NAVAL
POSTGRADUATE
SCHOOL

MONTEREY, CALIFORNIA

THESIS

DEFENDING IEEE 802.11-BASED NETWORKS AGAINST DENIAL OF SERVICE ATTACKS

by

Boon Hwee Tan

December 2003

Thesis Advisor:	William J. Ray
Second Reader:	Man-Tak Shing

Approved for public release; distribution is unlimited.

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE		Form Approved OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE December 2003	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE: DEFENDING IEEE 802.11-BASED NETWORKS AGAINST DENIAL OF SERVICE ATTACKS	5. FUNDING NUMBERS
6. AUTHOR(S) Boon Hwee Tan	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited	12b. DISTRIBUTION CODE

13. ABSTRACT (maximum 200 words)

The convenience of IEEE 802.11-based wireless access networks has led to widespread deployment in the consumer, industrial and military sectors. However, this use is predicated on an implicit assumption of confidentiality and availability. In addition to widely publicized security flaws in IEEE 802.11's basic confidentially mechanisms, the threats to network availability presents any equal, if not greater danger to users of IEEE 802.11-based networks. It has been successfully demonstrated that IEEE 802.11 is highly susceptible to malicious denial-of-service (DoS) attacks targeting its management and media access protocols.

Computer simulation models have proven to be effective tools in the study of cause and effect in numerous fields. This thesis involved the design and implementation of a IEEE 802.11-based simulation model using OMNeT++, to investigate the effects of different types of DoS attacks on a IEEE 802.11 network, and the effectiveness of corresponding countermeasures.

14. SUBJECT TERMS IEEE 802.11, WLAN, Wireless LAN, Protocol, Computer Security, Denial of Service, Simulation, OMNeT	15. NUMBER OF PAGES 133
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited.

DEFENDING IEEE 802.11-BASED NETWORKS AGAINST DENIAL OF SERVICE ATTACKS

Boon Hwee Tan
Major, Republic of Singapore Navy
B.ENG(EE), Nanyang Technological University, 1997

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 2003

Author: Boon Hwee Tan

Approved by: William J. Ray
Thesis Advisor

Man-Tak Shing
Second Reader

Peter Denning
Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

The convenience of IEEE 802.11-based wireless access networks has led to widespread deployment in the consumer, industrial and military sectors. However, this use is predicated on an implicit assumption of confidentiality and availability. In addition to widely publicized security flaws in IEEE 802.11's basic confidentially mechanisms, the threats to network availability presents any equal, if not greater danger to users of IEEE 802.11-based networks. It has been successfully demonstrated that IEEE 802.11 is highly susceptible to malicious denial-of-service (DoS) attacks targeting its management and media access protocols.

Computer simulation models have proven to be effective tools in the study of cause and effect in numerous fields. This thesis involved the design and implementation of a IEEE 802.11-based simulation model using OMNeT++, to investigate the effects of different types of DoS attacks on a IEEE 802.11 network, and the effectiveness of corresponding countermeasures.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

I. INTRODUCTION 1
A. BACKGROUND 1
B. OBJECTIVE 2

II. IEEE 802.11 STANDARD 3
A. IEEE 802.11 ARCHITECTURE 3
B. DISTRIBUTION SYSTEM 5
C. SERVICES 5
1. Station Services 6
2. Distribution Services 6
D. INTERACTION BETWEEN SOME SERVICES 7
E. MEDIUM ACCESS CONTROL 9
F. INTERFRAME SPACE (IFS) 10
G. DISTRIBUTED COORIDINATION FUNCTION 12
H. POINT COORDINATION FUNCTION 13

III. IEEE 802.11 MAC VULNERABILITIES 15
A. IDENTITY VULNERABILITIES 15
1. Deauthentication 15
2 Disassociation 16
B. MEDIA ACCESS VULNERABILITIES 18

IV. SIMULATION MODEL DEVELOPMENT 21
A. OMNET++ ENVIRONMENT 21
B. MODEL DESCRIPTION 22
C. KEY MAC FUNCTIONS 23

V. ATTACK STRATEGIES AND SIMULATION FINDINGS 25
A. ATTACK GENERES 25
1. Deauthentication Attacks 25
2. Disassociation Attacks 26
3. RTS Attacks 26
B. SIMULATION RESULTS OF ATTACK SCENARIOS 27
1. Baseline Scenario 27
2. Deauthentication Attack Scenario 28
3. Disassociation Attack Scenario 32
4. RTS Attack Scenario 35
C. SIMULATION RESULTS OF DEFENSE SCENARIOS 39
1. Defense Against Deauthentication and Disassociation Attacks 39
2. Defense Against RTS Attacks 44
D. SUMMARY OF EFFECTIVENESS OF IMPLEMENTED DEFENSES 47

VI. CONCLUSION 49
APPENDIX A. SOURCE CODE LIST OF SIMULATION MODEL 51
LIST OF REFERENCES 115
INITIAL DISTRIBUTION LIST 117

LIST OF FIGURES

Figure 1. Independent Basic Service Set (BSS) (After [4]) 4
Figure 2. Infrastructure BSS (After [4]) 4
Figure 3. Extended Service Set (ESS) (After [4]) 5
Figure 4. State Machine of Mobile Station (After [4]) 8
Figure 5. IEEE 802.11 MAC Architecture (After [4]) 10
Figure 6. IFS relationships (After [4]) 11
Figure 7. RTS / CTS / Data / ACK and NAV Setting (After [4]) 13
Figure 8. Graphical depiction of deauthentication attack (After [03]) 16
Figure 9. Graphical depiction of disassociation attack (After [3]) 17
Figure 10. Hierarchy of OMNeT++ Modules (After [6]) 21
Figure 11. Screen Shot of Implemented Model 22
Figure 12. RTS Attack 27
Figure 13. Simulation Results of Baseline Scenario 28
Figure 14. Simulation Results of Deauthentication Attack Scenario (Attack Cycle 3333 frames/sec) 29
Figure 15. Simulation Results of Deauthentication Attack Scenario (Attack Cycle 3636 frame/sec) 30
Figure 16. Simulation Results of Deauthentication Scenario (Attack Cycle 4000 frames/sec) 31
Figure 17. Simulation Results of Disassociation Scenario (Attack Cycle 3636 frames/sec) 33
Figure 18. Simulation Results of Disassociation Scenario (Attack Cycle 5000 frames/sec) 34
Figure 19. Simulation Results of RTS Attack Scenario (Attack Cycle 2000 frames/sec, Duration Field 310) 36
Figure 20. Simulation Results of RTS Attack Scenario (Attack Cycle 2000 frames/sec, Duration Field 350) 37
Figure 21. Simulation Results of RTS Attack Scenario (Attack Cycle 1000 frames/sec, Duration Field 850) 38
Figure 22. Simulation Results of Defense Against Deauthentication Scenario (Attack Cycle 4000 frames/sec, Timeout 500us) 40
Figure 23. Simulation Results of Defense Against Disassociation Scenario (Attack Cycle 5000 frames/sec, Timeout 500us) 41
Figure 24. Simulation Results of Defense Against Disassociation Scenario (Attack Cycle 6667 frames/sec, Timeout 500us) 43
Figure 25. Simulation Results of Defense Against RTS Attack Scenario (Attack Cycle 2000 frames/sec, Duration Field 350, Timeout 128us) 45
Figure 26. Simulation Results of Defense Against RTS Attack Scenario (Attack Cycle 1000 frames/sec, Duration Field 850, Timeout 128us) 46

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

Table 1. Average Values from Baseline Scenario 28
Table 2. Average Value from Deauthentication Scenario (Attack Cycle 3333 frames/sec)..... 29
Table 3. Average Value from Deauthentication Scenario (Attack Cycle 3636 frames/sec)..... 30
Table 4. Average Value from Deauthentication Scenario (Attack Cycle 4000 frames/sec)..... 31
Table 5. Average Value from Disassociation Scenario (Attack Cycle 3636 frames/sec)..... 33
Table 6. Average Value from Disassociation Scenario (Attack Cycle 5000 frames/sec)..... 35
Table 7. Average Value from RTS Attack Scenario (Attack Cycle 2000 frames/sec, Duration Field 310) 36
Table 8. Average Value from RTS Attack Scenario (Attack Cycle 2000 frames/sec, Duration Field 350) 37
Table 9. Average Values from RTS Attack Scenario (Attack Cycle 1000 frames/sec, Duration Field 850) 38
Table 10. Average Values From Defense Against Deauthentication Scenario (Attack Cycle 4000 frames/sec, Timeout 500us) 40
Table 11. Average Values From Defense Against Disassociation Scenario (Attack Cycle 5000 frames/sec, Timeout 500us) 42
Table 12. Average Values From Defense Against Disassociation Scenario (Attack Cycle 6667 frames/sec, Timeout 500us) 43
Table 13. Average Values From Defense Against RTS Attack Scenario (Attack Cycle 2000 frames/sec, Duration Field 350, Timeout 128us). 45
Table 14. Average Values From Defense Against RTS Attack Scenario (Attack Cycle 1000 frames/sec, Duration Field 350, Timeout 128us). 46
Table 15. Consolidated Data Rate Degradation..... 47

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank my thesis advisors Dr. William Ray and Dr. Man-Tak Shing for their assistance in making this thesis possible. A very special thanks to my wife Janet and son Koen for their love and encouragements.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. BACKGROUND

It is possible that in the future every laptop, handheld device, and desktop PC is connected wirelessly to the enterprise network. Today, enterprises are rapidly deploying wireless networks based on standards such as Institute of Electrical and Electronics Engineers (IEEE) 802.11b, which offers constant access to enterprise intranet, extranet, and Internet data and services. Compared to traditional wired networks, wireless networks offer mobility, giving users access to enterprise data anywhere and anytime; flexibility, reducing deployment and network reconfiguration costs; and convenience. These proven conveniences of wireless networks have led to widespread deployment in the consumer, industrial and military sectors.

However, there are some concerns associated with wireless networks. The greatest of which is security. Military and civil network managers must ensure that new vulnerabilities are not introduced to their enterprise network when a wireless LAN is deployed. At the same time, they must ensure that wireless transmissions are safe from eavesdropping, illegal data alteration and denial of service attacks.

A popular implementation of wireless networks is the IEEE 802.11 WLANs. The widespread deployment makes IEEE 802.11-based networks an attractive target for potential attackers. Indeed, research has demonstrated basic flaws in 802.11's encryption mechanisms [1] and authentication protocols [2] – ultimately leading to the creation of a series of protocol extensions and replacements (e.g., WPA, 802.11i, 802.1X) to address these problems. However, most of this work has focused primarily on the requirements of access control and confidentiality, rather than availability. Recent works [3] has highlighted the vulnerability of IEEE 802.11-based networks against attacks on the availability of the network.

B. OBJECTIVE

This thesis presents results from a simulation study evaluating the vulnerabilities of the IEEE 802.11 medium access control (MAC) protocol against malicious Denial of Service (DoS) attacks. Attacks targeting the identity vulnerability and virtual carrier sense mechanism are simulated and various countermeasures are subsequently implement to investigate their effectiveness against the corresponding attacks.

II. IEEE 802.11 STANDARD

A brief description of the IEEE 802.11 standard is provided in this chapter. This chapter focuses on the key areas of the IEEE 802.11 protocol that are pertinent to this work, and is not intended to provide a comprehensive overview of the capabilities of the protocol. Detailed descriptions of the complete standard can be found in [4] and [5], which provided the basis of this chapter.

A. IEEE 802.11 ARCHITECTURE

The architecture of the IEEE 802.11 WLAN is designed to support a network where most decision-making is distributed to the mobile stations. This architecture has several advantages, including being very tolerant of faults in all the WLAN equipment and eliminating any possible bottlenecks a centralized architecture would introduce. The architecture is very flexible, easily supporting both small, transient networks and large semi permanent or permanent networks.

The IEEE 802.11 WLAN architecture is built around a basic service set (BSS). The BSS consists of two or more wireless mobile stations, which can communicate with each other. There are two types of BSS. The first type being the independent BSS (IBSS), where mobile stations can communicate directly with each other (Figure 1). Mobile stations must be in communication range of each other in order to communicate directly, and there is no relay function in an IBSS.

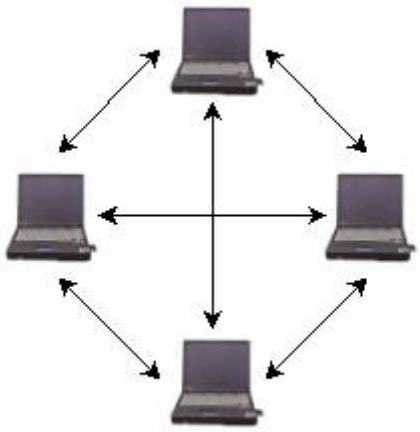

Figure 1. Independent Basic Service Set (BSS) (After [4])

The second type of BSS is the infrastructure BSS. The mobile stations communicate with each other through an Access Point (AP) (Figure 2). The AP can act as a bridge to connect the WLAN to a wired network. More importantly, the AP is the central node for all communications in the WLAN. All frames must be sent to the AP first, and the AP will retransmit the frame to intended destination mobile station in the BSS. This consumes twice as much bandwidth as compared to a one-link communication in an IBSS. However, the AP provides for traffic buffering if a destination station is not ready to receive the message.

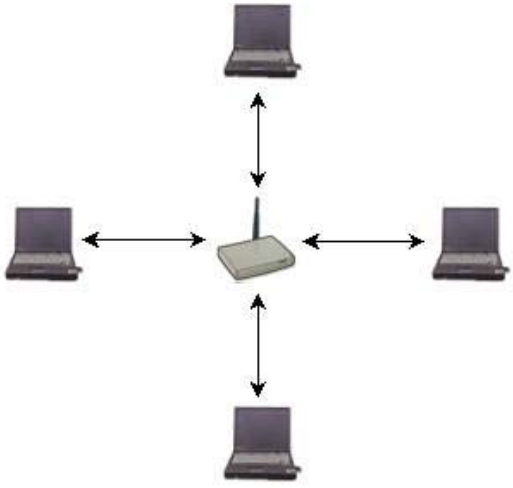

Figure 2. Infrastructure BSS (After [4])

A set of infrastructure BSS can be string together through the APs to form an Extended Service Set (ESS). This facilitates the movement of mobile stations from one BSS to another BSS within the ESS (Figure 3). The APs communicate among themselves to forward traffic between each BSS. The APs communicate via an abstract medium called the Distribution System (DS). The DS is the backbone of the WLAN and may be constructed of either wired or wireless networks.

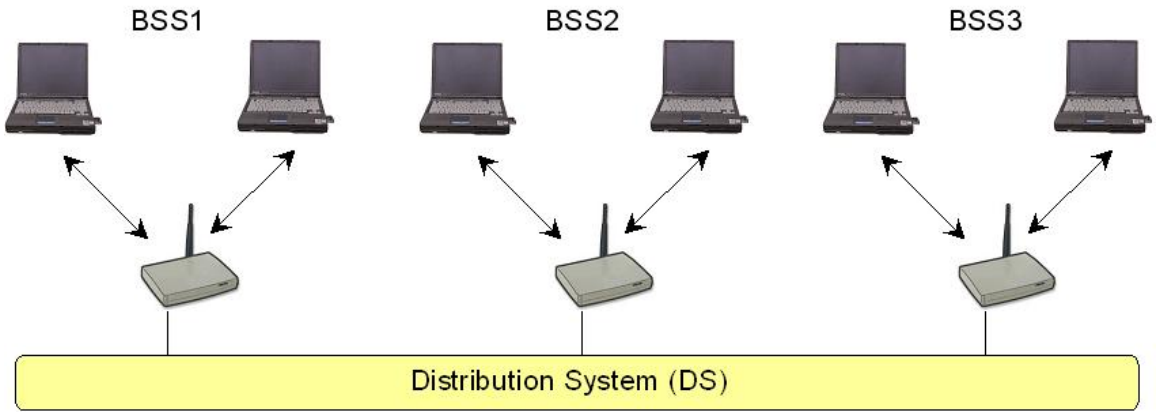

Figure 3. Extended Service Set (ESS) (After [4])

B. DISTRIBUTION SYSTEM

The DS is the mechanism by which one AP communicates with another to exchange frames for stations in their BSSes, forward frames to follow mobile stations from one BSS to another, and exchange frames with wired networks, if any. The DS is not necessarily a network, as long as the services it must provide exists. Thus the DS may be a wired network; e.g. 802.3 Ethernet, or a dedicated black box providing the required services.

C. SERVICES

The IEEE 802.11 defines a total of 9 types of services, divided into 2 groups; station services and distribution services.

1. Station Services

There are 4 station services; authentication, deauthentication, privacy and data delivery. These 4 services provide the WLAN with similar functions to those that are expected of a wired network. An authenticated mobile station is similar to a physically connected station in a wired network. Similarly, a deauthentication of a mobile station corresponds to disconnecting a station from the wired network. The authentication service is used to prove the identity of the mobile station to another. As such, only authenticated users are allowed to use the WLAN. The deauthentication service disconnects the mobile station from the WLAN and that station can no longer access service of the WLAN.

The privacy service of the IEEE 802.11 is designed to provide an equivalent level of protection for data traversing the WLAN as provided by a wired network. This service protects the data only as it traverses the wireless medium, and is not designed to provide end-to-end protection in a heterogeneous network.

The data transmission service is probably the most utilized service of the mobile station. This service is responsible for reliably delivering data frames from the Medium Access Control (MAC) of one mobile station to one or more other stations.

2. Distribution Services

The 5 distribution services of the protocol are association, reassociation, disassociation, distribution and integration. These distribution services function to allow mobile stations to roam freely within an ESS and to allow the IEEE 802.11 WLAN to be connected with a wired LAN infrastructure.

The association service is used to make a logical connection between a mobile station and an AP. This logical connection is necessary in order for the DS to know where and how to deliver data to the mobile station. The logical connection is also necessary for the AP to accept data frames from the mobile station and to allocate resources to support the mobile station.

The reassociation service is similar to the association service, with the exception that it includes information about the AP with which a mobile station has been previously associated. A mobile station will use the reassociation service repeatedly as it roams about the ESS, losing contact with the AP with which it has been associated with, and thus needing to be associated with a new AP as the station proceeds to within the new AP's coverage zone. By using reassociation service, a mobile station provides information to the new AP to which it will be associated, that allows that new AP to contact the AP with which the mobile station was previously associated, to obtain frames that may be waiting there for delivery to the mobile station.

The disassociation service is used either to force a mobile station to associate with an AP or for a mobile station to inform an AP that it no longer requires the services of the WLAN. An AP can uses this service if it wishes to inform mobile stations that it can no longer provide them with services to the WLAN. This can be due to network overloading, or network shutting down.

An AP determines how to deliver the frames it receives by using the distribution service. When a mobile station sends a frame to the AP for delivery to another station, the AP invokes the distribution service to determine if the frame should be forwarded to another AP for onward transmission or to retransmit the frame back into the AP's BSS.

The integration service functions to connect the IEEE 802.11 WLAN to other LANs. This connection may include one or more wired LANs, or other IEEE 802.11 WLANS. This service translates IEEE 802.11 frames to frames that may traverse other networks, and vice versa. This service is similar to the service provided by multi protocol enabled router in the Internet.

D. INTERACTION BETWEEN SOME SERVICES

The authentication / deauthentication services and association/ disassociation services are used by the mobile stations to maintain two independent variables. These two variables; authentication state and association

state, are used in a simple state machine, which determines the order in which certain services must be invoked and when a station may begin using the data delivery service. A station may be authenticated with many different stations or APs simultaneously. However, it may only be associated with one station or AP at a time.

Stations always begin in state 1, when it is neither authenticated nor associated with any other station or AP. When in state 1, the mobile station has only access to those services that allow the station to discover the WLAN and the authentication service. If the station is successful in authenticating with another station or AP, the station will transit to state 2. (see Figure 4).

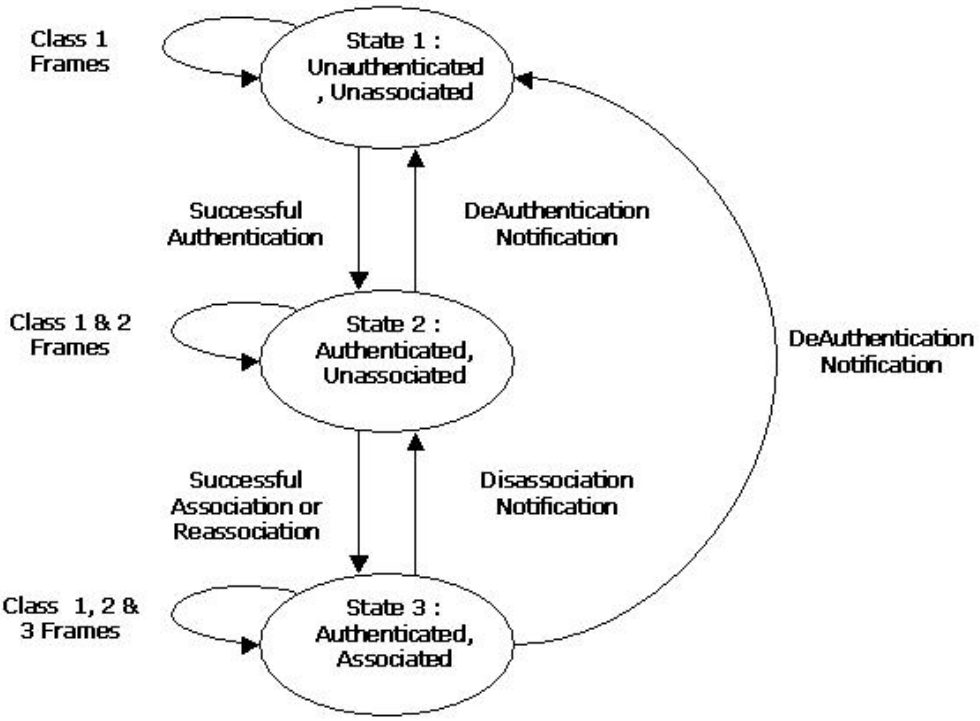

Figure 4. State Machine of Mobile Station (After [4])

When a mobile station transits from state 1 to state 2; i.e. it successfully authenticates with an AP or station, the authentication variable is set to true, and the association variable remains false. In state 2, the mobile station can access

additional services that allow the station to associate, reassociate and disassociate with another station or AP. If the mobile station is not successful in becoming associated, the mobile station will remain in state 2. When in state 2, the mobile station receives a deauthentication notification, the mobile station will return to state 1, and the authentication variable will be set to false.

When in state 2, the mobile station successfully associates with another station or AP, its association variable will be set to true and the station will make a transition to state 3. In state 3, a mobile station is allowed to use all frame types and the data delivery service. The mobile station will remain in state 3, until it receives either a disassociation or deauthentication notification.

A station must react to frames it receives in each of the states, even those that are disallowed for a particular state. For example, a mobile station in state 1 receives an association frame from another station, the mobile station must respond with a deauthentication frame. This mandatory response forces the station that sent the disallowed frames to make a transition to the proper state in the state diagram and thus allowing it to proceed properly towards state 3.

E. MEDIUM ACCESS CONTROL

The IEEE 802.11 medium access control (MAC) supplies the functionality required to provide a reliable delivery mechanism for user data over noisy, unreliable wireless media. Although the data delivery itself is based on an asynchronous, best-effort, connectionless delivery of MAC layer data, the frame exchange protocol at the MAC level does significantly improves on the reliability of data delivery over wireless media [5]. It does this while also providing advance LAN services, equal to or beyond those of existing wired LANs.

The fundamental access method of IEEE 802.11 MAC is a Distributed Coordination Function (DCF) that provides controlled access method to the shared wireless medium, also known as Carrier-Sense Multiple Access with Collision Avoidance (CSMA/CA). CSMA/CA is similar to the collision detection access method deployed by IEEE 802.3 Ethernet LANs, but due to the wireless

media, it is not feasible for collision detection in the media, as such, an avoidance strategy is adopted.

The third function of the IEEE 802.11 MAC is to protect the data that it delivers. As it is difficult to contain a WLAN within any physical perimeters, the IEEE 802.11 MAC provides a privacy service; Wired Equivalent Privacy (WEP), which encrypt the data with RC4 cryptographic process before sending it over the wireless medium.

Apart from DCF, another method of media access control featured in the IEEE 802.11 protocol is the Point Coordination Function (PCF). Although DCF is the primary access control method used, it can also co-exist with PCF. Figure 5 illustrates the MAC Sub layer Architecture that allows for the coexistence of both functions.

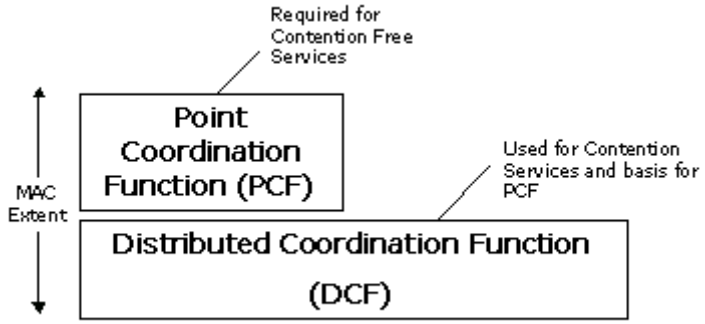

Figure 5. IEEE 802.11 MAC Architecture (After [4])

DCF is used in both independent and infrastructure networks; whereas, PCF is an optional access method and used only in infrastructure network configurations. In the infrastructure network, a point coordinator (PC) controls access to the medium permitting the DCF and PCF to coexist.

F. INTERFRAME SPACE (IFS)

A station needs to listen to the medium for a period of time before deciding if the medium is carrying any transmission. If the station does not detect any transmission during that period of time, then the station will determine that the

medium is free from transmission. The IEEE 802.11 MAC recognizes four such timing periods, known as Interframe spacing (IFS). The four different types of IFS determine the priorities of stations in accessing the medium. The first type is Short IFS (SIFS), used in sending an acknowledgement, Clear To Send (CTS) frames, and the second or subsequent frames of a fragment burst. During the contention-free period (CFP), a station also uses SIFS when it responds to a poll while a point coordinator (PC), which coordinates the communication in the WLAN, may use SIFS for any type of frame. A SIFS is the shortest IFS; consequently, provides a station with the highest priority in gaining access to the medium.

The second type of IFS is the Priority IFS (PIFS). Except when responding to a poll by PC, a station will use PIFS during the CFP for all other purposes under PCF.

The third type is a Distributed IFS (DIFS), which is used under the DCF. DIFS is the longest interframe space. Hence, a station waiting a DIFS period has the lowest priority. A point coordinator is guaranteed to gain and maintain control of the medium to start the CFP by employing PIFS instead of DIFS.

The fourth type of IFS is an Extended IFS (EIFS), used when the first attempt to transmit a frame has failed. The EIFS is shorter than DIFS, because a retransmission has higher priority than a normal transmission. Figure 6 illustrates the relationships between the IFS.

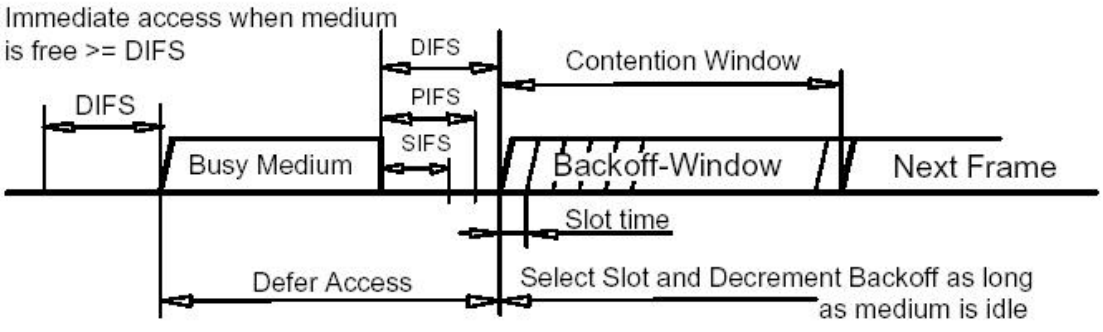

Figure 6. IFS relationships (After [4])

The 4 timing intervals are used to implement the two coordination functions; DCF and PCF, of the IEEE 802.11 protocol.

G. DISTRIBUTED COORIDINATION FUNCTION

DCF is mandatory and based on the CSMA/CA protocol. With DCF, 802.11 stations contend for access and attempt to send frames when there is no other station transmitting. If another station is sending a frame, other stations are expected to wait until the channel is free before attempting to transmit their data.

As a condition to accessing the medium, the MAC Layer checks the value of its network allocation vector (NAV), which is a counter resident at each station that represents the amount of time that the previous frame needs to send its frame. The NAV must be zero before a station can attempt to send a frame. Prior to transmitting a frame, a station calculates the amount of time necessary to send the frame based on the frame's length and data rate. The station places a value representing this time in the duration field in the header of the frame. The IEEE 802.11 defines the Request-To-Send (RTS) frames and Clear-To-Send (CTS) frames for the purpose of medium reservation by stations. When stations receive the RTS or CTS frames, they examine this duration field value and use it as the basis for setting their corresponding NAVs. This process reserves the medium for the sending station. Figure 7 illustrates the process of using RTS, CTS frames to reserve the medium.

An important aspect of the DCF is a random back off timer that a station uses if it detects a busy medium. If the channel is in use, the station must wait a random period of time before attempting to access the medium again. This ensures that multiple stations wanting to send data don't all transmit at the same time. The random delay causes stations to wait different periods of time and avoids all of them sensing the medium at exactly the same time, finding the channel idle, transmitting, and colliding with each other. The back off timer significantly reduces the number of collisions and corresponding retransmissions, especially when the number of active users increases.

With WLANs, a transmitting station can't listen for collisions while sending data, mainly because the station can't have it's receiver on while transmitting the frame. As a result, the receiving station needs to send an acknowledgement (ACK) if it detects no errors in the received frame. If the sending station doesn't receive an ACK after a specified period of time, the sending station will assume that there was a collision or medium interference, and retransmit the frame.

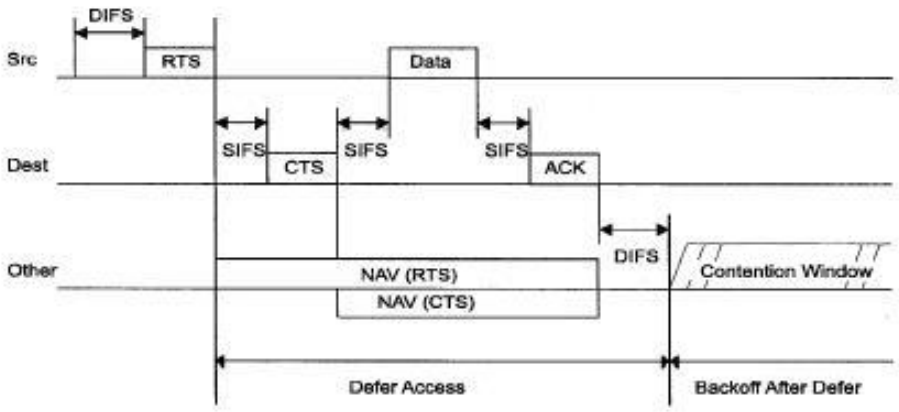

Figure 7. RTS / CTS / Data / ACK and NAV Setting (After [4])

H. POINT COORDINATION FUNCTION

For supporting time-bounded delivery of data frames, the IEEE 802.11 protocol defines the optional PCF where the access point grants access to an individual station to the medium by polling the station during the contention free period. Stations can't transmit frames unless the access point polls them first. The period of time for PCF-based data traffic; if enabled, occurs alternately between contention periods.

The access point polls stations according to a polling list, then switches to a contention period when stations use DCF. This process enables support for both synchronous and asynchronous modes of operation.

However, no known wireless Network Interface Cards (NICs) or AP on the market today, however, implement PCF.

THIS PAGE INTENTIONALLY LEFT BLANK

III. IEEE 802.11 MAC VULNERABILITIES

The 802.11 MAC layer incorporates additional functionality designed to address problems of the wireless medium. These functions include the ability to discover networks, join and leave networks, and coordinate access to the wireless medium. The vulnerabilities discussed in this chapter are a consequent of these additional functionalities and can be broadly placed into two categories: identity and media-access control [3].

A. IDENTITY VULNERABILITIES

Identity vulnerabilities arise from the implicit trust IEEE 802.11 networks place in a transmitting station's address. APs and Mobile Stations identify themselves in an IEEE 802.11 network by their unique 48 bit MAC addresses. These addresses are found in the unencrypted portion of IEEE 802.11 communication frames. There is no mechanism in the protocol to verify the authenticity of the self-reported MAC address. As such, an attacker may "spoof" any node of his choice and request MAC-layer services on the victim's behalf. By invoking certain MAC layer services, an attack can force any node to leave the network involuntarily.

1. Deauthentication

In infrastructure mode, all mobile stations must communicate through an AP. In order to communicate with an AP, a mobile station must first authenticate itself with the AP. Part of the authentication framework is a message that allows mobile clients and access points to explicitly request deauthentication from one another. The deauthentication message is not authenticated using any keying material. As such, an attacker may either spoof the access point or the mobile station, and direct the deauthentication message to the other party (see Figure 8).

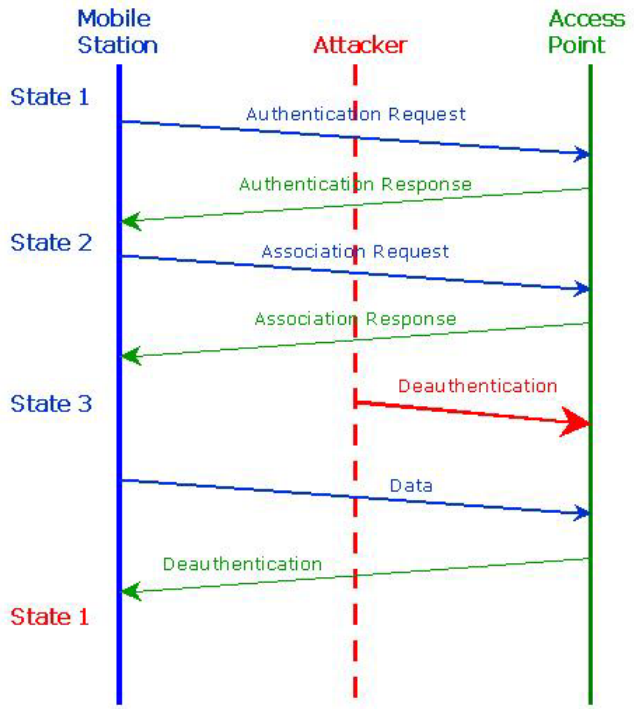

Figure 8. Graphical depiction of deauthentication attack (After [03])

The victim mobile station, upon receiving the deauthentication message, must deauthenticate itself from the AP. This response is mandated by the IEEE 802.11 protocol. The victim mobile station will consequently attempt to reauthenticate with an AP, since it still has a desire to join the network. The time needed to rejoin a network is a function of how aggressive the mobile station attempts to rejoin the network. As a condition for the attack to be successful, the attacker needs to repeatedly deauthenticate the victim client, each time the client attempts to rejoin the network. This form of deauthentication attack provides the attacker with the flexibility of denying access to a mobile station of choice, or rate limit the victim's access.

2 Disassociation

A very similar vulnerability to the authentication protocol is the association protocol. A mobile station can be authenticated with multiple APs, but it must be associated with only one AP in order to use the data delivery service of the

network. An attacker can similarly exploit the unauthenticated dissociation message to deny or degrade a victim mobile station's access to the network's services (see Figure 9).

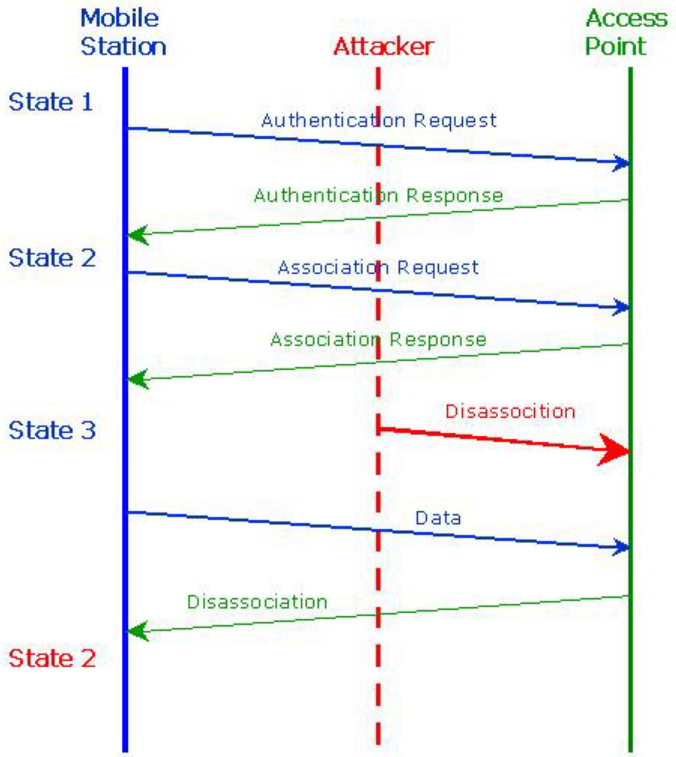

Figure 9. Graphical depiction of disassociation attack (After [3])

Referring to Figure 4, it can be observed that a deauthentication attack will always return the victim station to state 1, the lowest possible state of communication. In contrast, a disassociation attack will only return the victim to state 2. A mobile station must be in state 3 in order to use the data delivery service of the network. As such, the deauthentication attack is more efficient than the disassociation attack; as the victim mobile station needs to transit more states to attain state 3, comparing with a victim of a disassociation attack. This greater efficiency of attack can translate to lesser work required of the attacker in order to successfully mount the attack.

B. MEDIA ACCESS VULNERABILITIES

Due to the stochastic nature of wireless communications, it is not possible to implement perfect collision detection mechanisms in IEEE 802.11 networks. As such, the protocol use a combination of physical carrier sense mechanism and virtual carrier sense mechanism to control access to the wireless medium, in order to avoid collisions. However, these two mechanisms inadvertently provide opportunities for an attacker to conduct DoS attacks on the mobile stations.

In order to prioritize access to the wireless medium, the IEEE 802.11 protocol has defined 4 IFS of different timing periods. The shortest IFS is the SIFS, which has a period of 20 microseconds. Before any mobile station can commence to use the wireless medium, each station must listen to the medium for one IFS period; the specific type of IFS depends on the state of communication of the particular mobile station. If there were no transmissions detected during the IFS period, the mobile station will wait a short random period before commencing to use the medium. The random period is imposed on the mobile stations to prevent multiple stations from accessing the medium simultaneously. If there is a collision detected during the transmission of a station's data, the station will suspend its transmission and back off for a period of time determined by an exponential back off algorithm, before trying to access the medium again. An attacker can exploit the need for all stations to wait for at least SIFS period of time, before any station can access the medium, to conduct a DoS attack on all stations in the network. The attacker sends a random frame before the end of every SIFS cycle repeatedly. Upon detecting the random frame, all mobile stations will back off from accessing the medium and will be unable to send out their legitimate data or requests. This kind of attack demands a very high workload on the attacker as the attacker is expected to send out a frame in less than 20 microseconds, repeatedly [3].

Another avenue of attack targets the virtual carrier sense mechanism of the protocol. Each mobile station updates their respective NAV value when they receive frames from other nodes. The NAV value indicates to the station the amount of time the medium is reserved; the station must wait until its NAV value

reaches nil before the station tries to access the medium. This feature is used primarily in the RTS/CTS handshake for medium reservation by mobile stations.

An attacker can send a RTS frame with an exceptionally high value in the duration value field. The destination station is mandated to respond with a CTS frame with the duration value updated to account for time elapsed during the RTS/CTS exchange. Other mobile stations, upon receiving the CTS frame will update their respective NAV with the exceptionally large duration value. This causes these stations to defer their access to the medium by a much long period than necessary. The maximum duration value that is allowable under the IEEE 802.11 protocol is 32767, which roughly translates to about 32 milliseconds. A major advantage of this form of attack is that the attacker can direct the RTS frame to a station that has high output power. The high output power station is able to propagate the CTS response frame to a wider area, and thus increases the probability that more mobile stations can be affected by the attack. An example of a high power station is an AP.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. SIMULATION MODEL DEVELOPMENT

A. OMNET++ ENVIRONMENT

OMNeT++ is the acronym for Objective Modular Network Testbed in C++. It is an object-oriented modular discrete event simulator [6]. The simulator can be used for:

- Communication protocols modeling
- Computer networks modeling (traffic modeling etc.)
- Modeling multi-processors and distributed systems
- Modeling any other system where the discrete event approach is suitable.

An OMNeT++ model comprises of hierarchically nested modules. There is no limit to the depth of module nesting. This facilitates the implementation of the logical structure of complex systems, which often entails many levels of abstraction. Figure 10 illustrates the nomenclature of the hierarchy module nesting in OMNeT++. The hierarchically modules are constructed using a graphical interface called the Graphical Network Description (GNED) editor. This graphical tool allows users to have an overview of the logical implementation of the model; including the communication channels between the modules and the hierarchical relationship between modules. This visualization tool reduces the learning curve for OMNeT++, and allows users to build models quickly and accurately.

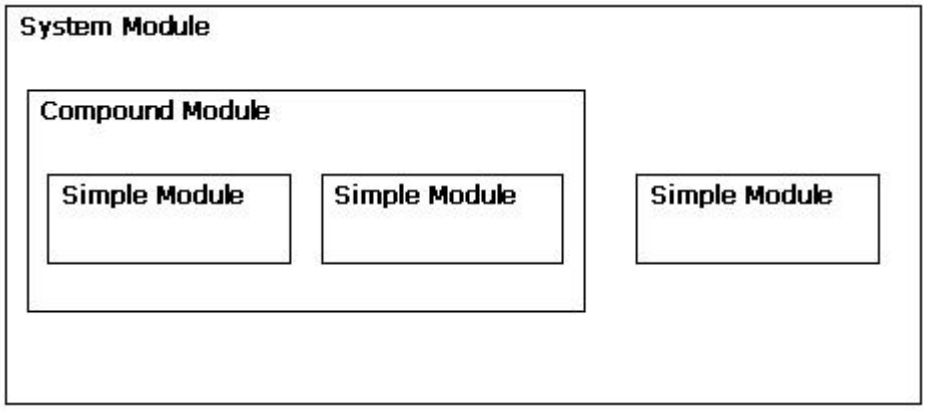

Figure 10. Hierarchy of OMNeT++ Modules (After [6])

B. MODEL DESCRIPTION

The IEEE 802.11 model is implemented using OMNeT++ in the Microsoft Visual C++ environment. The model comprise of only the MAC layer implementation, the physical layer is not explicitly modeled as the effects of the different transmission times due to different mediums can be simulated by using different timing values in the MAC layer. The model comprises of an AP, a medium module, an attacker module, and one or more mobile stations, as defined by the user. Figure provides a snap shot of the implemented model on the graphical display of OMNeT++. In this instance, there are 4 mobile stations.

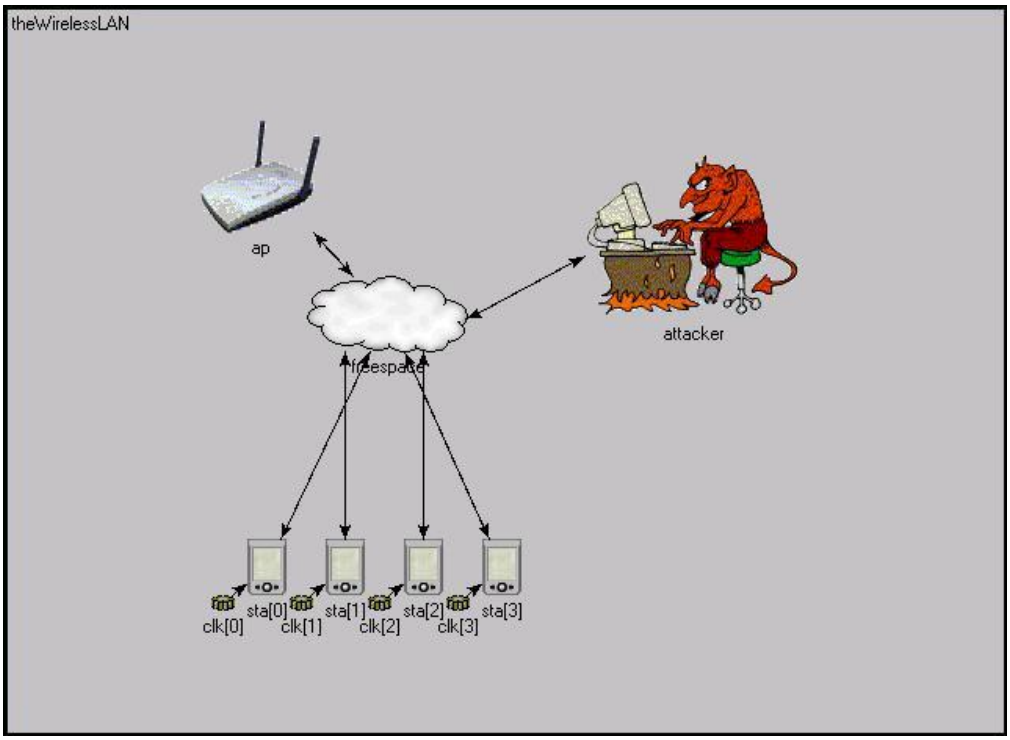

Figure 11. Screen Shot of Implemented Model

Although OMNeT++ is a discrete event simulator, a clock drive approach was preferred over an event driven approach for the IEEE 802.11 model. This is due to the need for mobile stations to listen to the network regularly to implement

the physical and virtual carrier sense mechanisms. A clock driven model would account for activities in every cycle and thus facilitates the accounting of NAV decrements and timeout triggers; e.g. DIFS timeout for medium reservation.

C. KEY MAC FUNCTIONS

The IEEE 802.11 MAC comprise of many functions and services necessary to ensure reliable communication. These services include security services, MSDU ordering, data services, distribution services, etc. The identity and media access vulnerabilities are localized in some of these services; as such the model does not implement all functionalities of the protocol. For the purpose of studying the effects of vulnerabilities, the following key functions are implemented:

- Authentication, Association, Deauthentication and Disassociation Services
- Virtual carrier sense mechanism
- Distributed Coordination Function

It was envisaged that the non-inclusion of the other functionalities of the protocol would have significant impact on the results of the simulation. This was because these additional functionalities are not affected by the attack strategies adopted; these functionalities do not interact with the spoofed frames generated by an attacker station during the conduct of an attack.

THIS PAGE INTENTIONALLY LEFT BLANK

V. ATTACK STRATEGIES AND SIMULATION FINDINGS

A. ATTACK GENERES

An incremental approach was adopted in the simulation of attacks and defenses of the IEEE 802.11 model. An initial unprotected model is subjected three genres of attacks. Thereafter, the model is progressively hardened against each kind of attack, and the effectiveness of the defensive measures are evaluated. In order for all attacks to be effective, they must be repeated periodically; i.e. attacker station must periodically transmit the spoofed frame on the network.

The cyclical period of transmitting spoof frames or conducting attacks is termed as the attack period. Alternatively the number of times a spoofed frame is transmitted or an attack is conducted over a 1 second time interval is termed as the attack cycle.

The three genres of attacks under study are as follows:

1. Deauthentication Attacks

The management frames of the IEEE 802.11 protocol are not authenticate by the receiver, this allows an attack to spoof as a legitimate station in the network to send management frames to any other stations in the network. The attacker spoofs a mobile station to request to deauthenticate from the currently associated AP. The AP upon receiving the request must respond with a deauthentication frame with its destination as the spoofed mobile station (see Figure 8). The spoofed mobile station, after receiving the deauthentication frame from the AP, will set its authentication variable to false. This causes the victim mobile station to drop to state 1 of communication state with the AP (see Figure 4).

As the victim mobile station retains the desire to continue to communicate with the network, the victim mobile station will reinitiate the authentication and association with the same AP or another AP within range. In doing so, the victim

station needs obey all timing requirements of the DCF in order to access the medium to rejoin the network. The attacker needs to repeatedly spoof the victim mobile station in order to continuously deny the victim mobile station access to the network, or to degrade the station's access rate. This form of attack affords the attacker with the ability to select a particular mobile station within the network for DoS attack.

2. Disassociation Attacks

This attack is very similar to the deauthentication attacks. Instead of requesting for deauthentication, the attacker requests disassociation from the currently associated AP, on behalf of the victim mobile station (see Figure 9). The victim mobile station, upon receiving the disassociation notification from the AP, will drop to state 2 of communication state with the AP (see Figure 4).

The victim mobile station will attempt to reassociate with the same AP or another AP that may be within range. Similarly the attacker needs to repeatedly requests disassociation from the AP in order for this form of attack to be effective, and the attacker has the ability to direct the attack on a victim station of choice.

3. RTS Attacks

This attack exploits the implicit trust placed on the MAC address of stations participating in a RTS/CTS handshake for medium reservation. An attacker can spoof the request for reservation of the medium on behalf of a mobile station by sending an RTS frame with the duration value set to an arbitrary value. The AP, upon receiving the request, will respond with a CTS frame to all stations within its transmission range. The CTS frame's duration value will be that of the initial value set in the RTS frame, less the time elapsed during the handshake. All stations receiving the CTS frame will update their respective NAV value, and will not attempt to access the medium until their NAV reaches zero (see Figure 12).

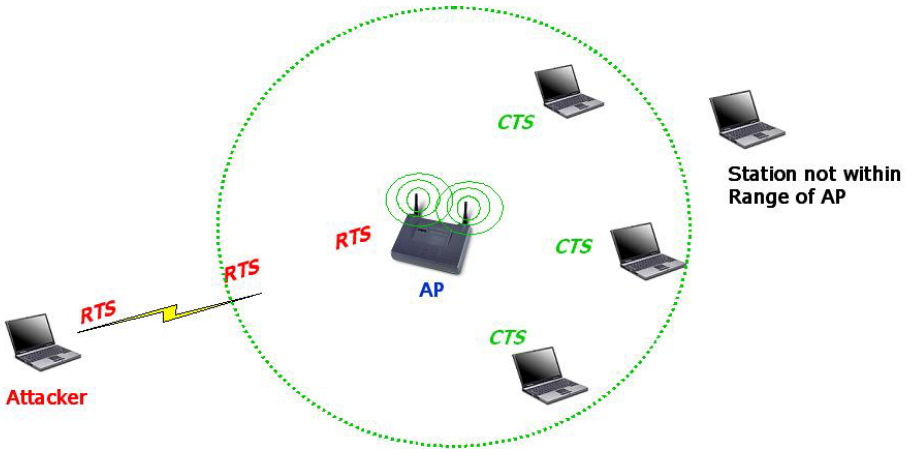

Figure 12. RTS Attack

The attacker can choose to set a high value in the duration field of the RTS frame, and thus denying other legitimate stations from assessing the medium for extended periods of time. This form of attack is non-discriminating as it affects all mobile stations within the AP transmission range, and it needs to be repeated periodically to sustain the denial of service or degradation effects.

B. SIMULATION RESULTS OF ATTACK SCENARIOS

All simulation scenarios are set up with 1 AP, 1 attacker station and 4 mobiles stations as depicted in Figure 11. The scenarios are each configured with different combinations of attacker station with different attack strategies and mobile stations with various defense enhancements.

1. Baseline Scenario

The mobile stations are implemented as described by the IEEE 802.11 protocol, without any additional protection enhancements to defend against attacks. The transmission rates of each of the 4 mobile stations over a 10 second time window are captured for analysis. Figure 13 and Table 1 captures the results from the simulation of the system in a benign environment.

The results from the simulation indicate healthy contention and access to the medium by all 4 mobile stations. This is inferred from the small differences of average data rates of all stations.

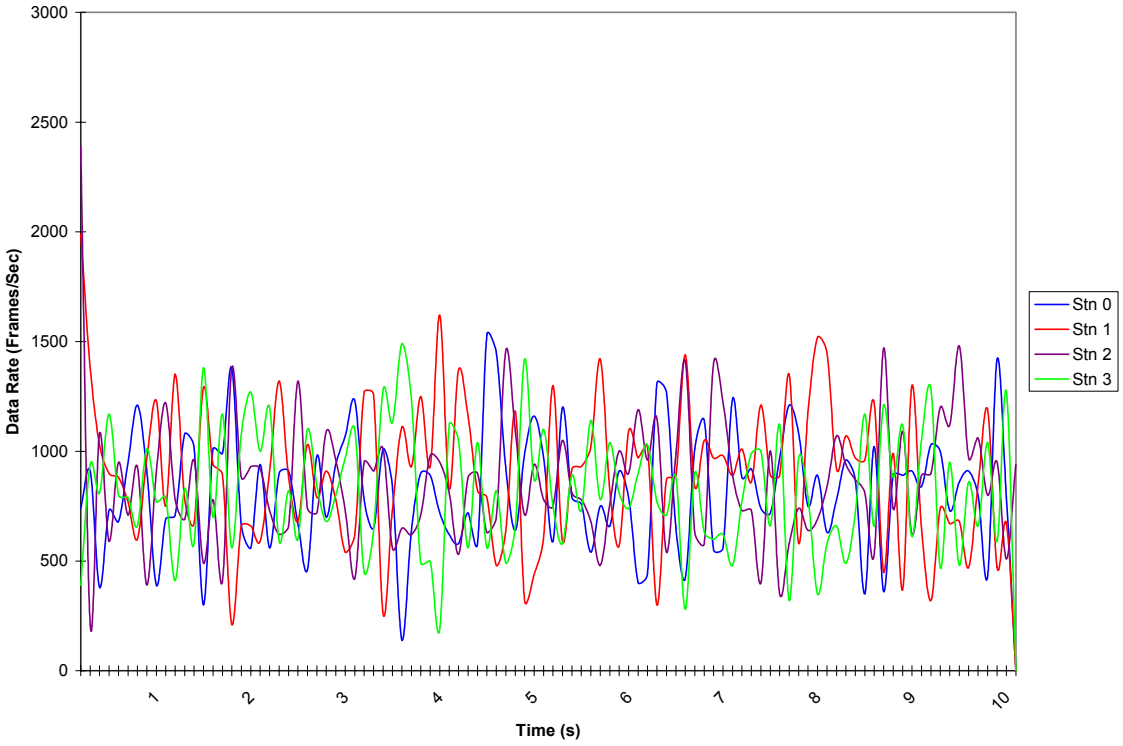

Figure 13. Simulation Results of Baseline Scenario

Simulation Time	Average Data Rate (Frames / sec)			
	Station 0	Station 1	Station 2	Station 3
0 - 3 sec	807.93	909.66	867.93	852.07
3 - 6 sec	831.67	909.33	813.33	855.33
6 - 10 sec	820.00	880.98	893.17	768.78

Table 1. Average Values from Baseline Scenario

2. Deauthentication Attack Scenario

The attacker station is configured to carry out deauthentication attacks periodically over the 3rd to 6th second interval of the simulation. Attacker stations conducting deauthentication attacks have the ability to choose victim stations. In

this series of attacks, the chosen victim is Station 1. The attack cycle is varied to demonstrate its effects on the effectiveness of this attack strategy.

Figure 14 and Table 2 captures the results of the simulation run with attack cycle set at 3333 frames/sec (attack period 300us).

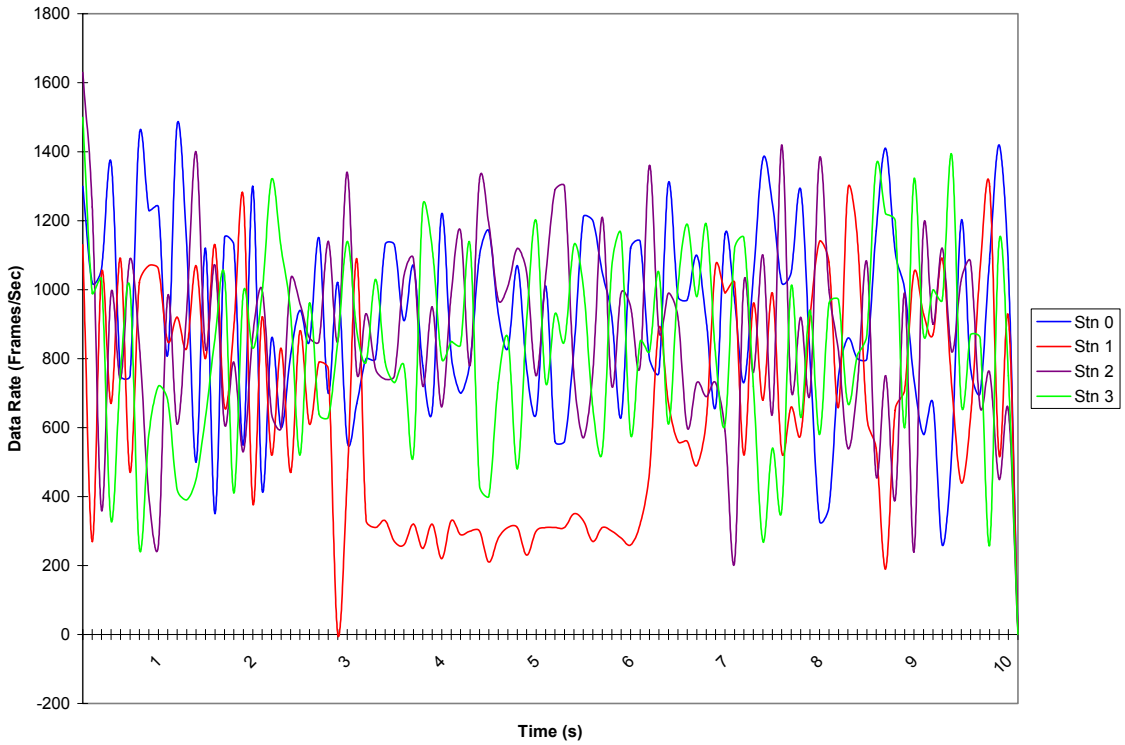

Figure 14. Simulation Results of Deauthentication Attack Scenario (Attack Cycle 3333 frames/sec)

Simulation Time	Average Data Rate (Frames / sec)			
	Station 0	Station 1	Station 2	Station 3
0 - 3 sec	951.72	788.62	877.59	791.72
3 - 6 sec	902.00	**319.67**	944.67	838.67
6 - 10 sec	900.98	757.80	799.27	858.54

Table 2. Average Value from Deauthentication Scenario (Attack Cycle 3333 frames/sec)

It was observed from the results of the simulation that the deauthentication attack with attack cycle of 3333 frames per second has the ability to degrade the

data rate of the victim station by about 60%. Although the victim station is still able to access the network, it suffers from the degradation effects of the attack.

The attack cycle is increased to 3636 frames per second (attack period 275us) during the attack interval from the 3rd second to 6th second of the simulation run. The victim station is Station 1. The results of the simulation are captured in Figure 15 and Table 3.

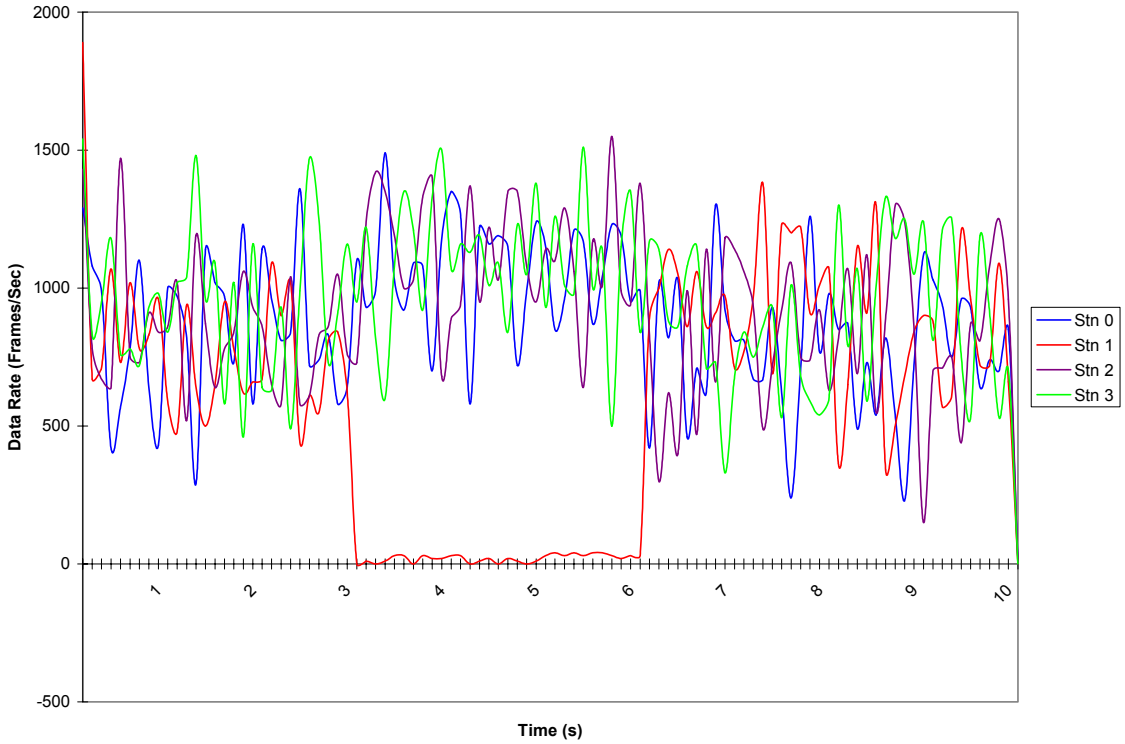

Figure 15. Simulation Results of Deauthentication Attack Scenario (Attack Cycle 3636 frame/sec)

Simulation Time	Average Data Rate (Frames / sec)			
	Station 0	Station 1	Station 2	Station 3
0 - 3 sec	850.69	793.79	853.45	948.97
3 - 6 sec	1066.67	20.33	1113.00	1098.67
6 - 10 sec	761.22	853.17	820.00	868.05

Table 3. Average Value from Deauthentication Scenario (Attack Cycle 3636 frames/sec)

It was observed from both Figure 15 and Table 3 that the victim station is almost totally denied from access to the network. Station 1 is only able to maintain a data rate of about 20 frames/sec during the period when the attacker station was conducting the attack. This victim station suffers a data rate degradation of about 97%.

The attack cycle is further increased to 4000 attacks per second (attack period 250us) and the results are captured in Figure 16 and Table 4.

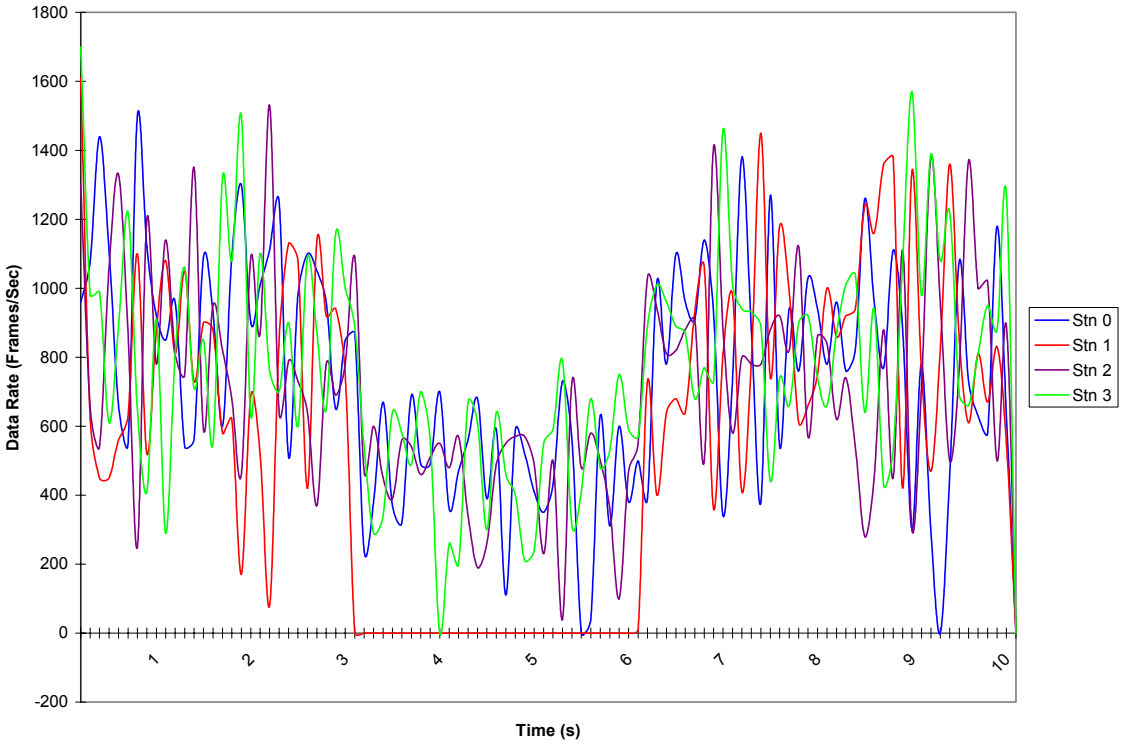

Figure 16. Simulation Results of Deauthentication Scenario (Attack Cycle 4000 frames/sec)

Simulation Time	Average Data Rate (Frames / sec)			
	Station 0	Station 1	Station 2	Station 3
0 - 3 sec	952.76	764.14	842.76	896.21
3 - 6 sec	**464.33**	**0.00**	**470.67**	**490.00**
6 - 10 sec	777.80	809.02	784.15	870.73

Table 4. Average Value from Deauthentication Scenario (Attack Cycle 4000 frames/sec)

It was observed that the victim station was unable to transmit any data during the period when the attacker station was conducting the attack. It was also observed that other stations that are not the target of the attack, also suffered from a degradation of about 50% in data rate during the same period. This could be attributed to the aggressive effort by Station 1, in attempting to reauthenticate with the AP each time it is deauthenticate with the AP. The aggressive and frequent transmission of Authentication Request frames by Station 1 causes the other mobile stations to defer their access to the medium for data transmission. This results in an overall reduction of data rate by the stations that are not the target of the attack.

3. Disassociation Attack Scenario

This series of scenarios are similar to the Deauthentication attack simulation run, except that the attacker station was configured to carry out disassociation attacks. Station 1 is the target victim station for all disassociation attacks during the 3rd to 6th second interval of the simulation.

A simulation run with disassociation attacks at 3636 frames/sec (attack period 275us) was conducted. The results of the simulation run are captured in Figure 17 and Table 5.

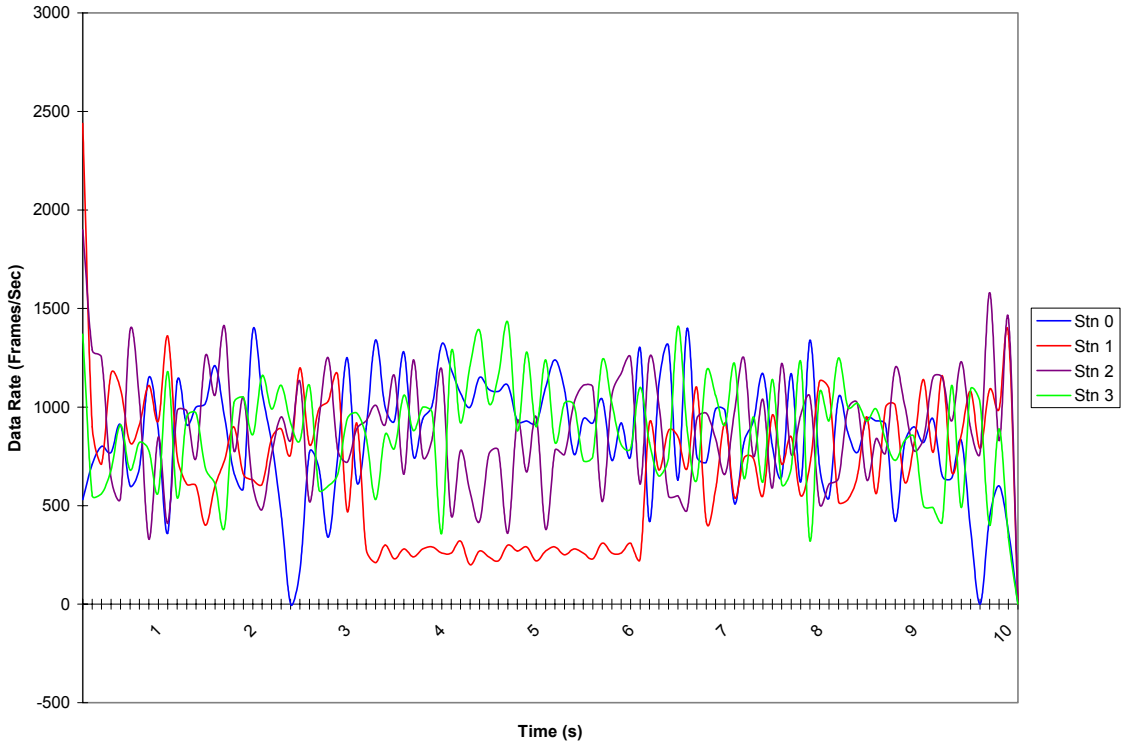

Figure 17. Simulation Results of Disassociation Scenario (Attack Cycle 3636 frames/sec)

Simulation Time	Average Data Rate (Frames / sec)			
	Station 0	Station 1	Station 2	Station 3
0 - 3 sec	778.97	899.31	926.21	830.34
3 - 6 sec	1001.33	**286.67**	846.67	972.33
6 - 10 sec	784.15	790.49	884.88	828.78

Table 5. Average Value from Disassociation Scenario (Attack Cycle 3636 frames/sec)

It was observed that the victim station's data rate was reduced by about 68% during the attack interval. Comparing this result with a deauthentication attack with the same attack cycle (Figure 15 and Table 3), where the victim station suffers a data rate degradation of about 97% due to the deauthentication attack. The greater degradation effect on the victim station's data rate

demonstrates the greater effectiveness of the deauthentication attack compared with the disassociation attack.

The basis for the greater effectiveness of deauthentication attacks over disassociation attacks can be inferred from the communication state machine of mobile stations (see Figure 4). It is observed that a deauthentication attack triggers a two state change to the Communication State of the victim mobile station, as opposed to a one state change arising from a disassociation attack. The victim mobile station can only transmit data frames only when it is in State 3 of the Communication State. As such, a deauthenticated victim station needs additional time and bandwidth to reestablish to State 3 before it can transmit data frames, as compared to a disassociated victim station.

The attack cycle is increase to 5000 frames/sec (attack period 200us). The results are captured in Figure 18 and Table 6.

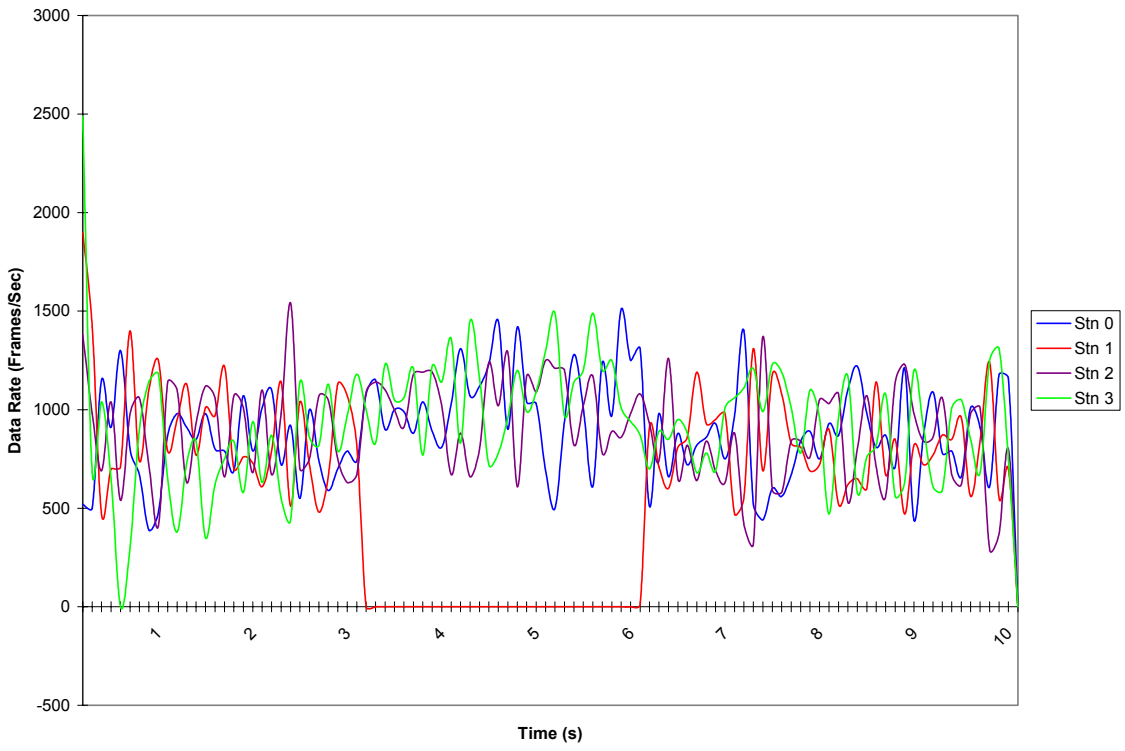

Figure 18. Simulation Results of Disassociation Scenario (Attack Cycle 5000 frames/sec)

Simulation Time	Average Data Rate (Frames / sec)			
	Station 0	Station 1	Station 2	Station 3
0 - 3 sec	812.76	926.55	911.72	802.07
3 - 6 sec	1036.00	**27.33**	1003.33	1107.33
6 - 10 sec	835.61	770.24	794.15	882.44

Table 6. Average Value from Disassociation Scenario (Attack Cycle 5000 frames/sec)

Comparing the results in Figure 18, Table 6 and those of deauthentication attack with attack cycle of 3636 frames/sec (Figure 15, Table 3), both victim stations suffer about 97% degradation in data rate. However, the disassociation attack was conducted at a higher attack cycle at 5000 frames/second. This reinforces the conclusion that the deauthentication attack is superior than the disassociation attack in attaining the objectives of a DoS attack.

4. RTS Attack Scenario

For this series of simulations, the attacker station was configured to carry out RTS attacks periodically over the 3rd to 6th second interval of the simulation. The attacker station conducting the attacks does not have the ability to target and choose victim stations for this attack. This form of attack affects all mobile station within the transmission range of the attacker station or the AP that was unwittedly used to collaborate the attack. Apart from the attack cycle that was varied, the duration field of the RTS frame was also varied to demonstrate its effects on the effectiveness of the RTS attack strategy.

Figure 19 and Table 7 captures the results of the simulation run with attack cycle set at 2000 frames/sec (attack period 500us) and duration field set at 310.

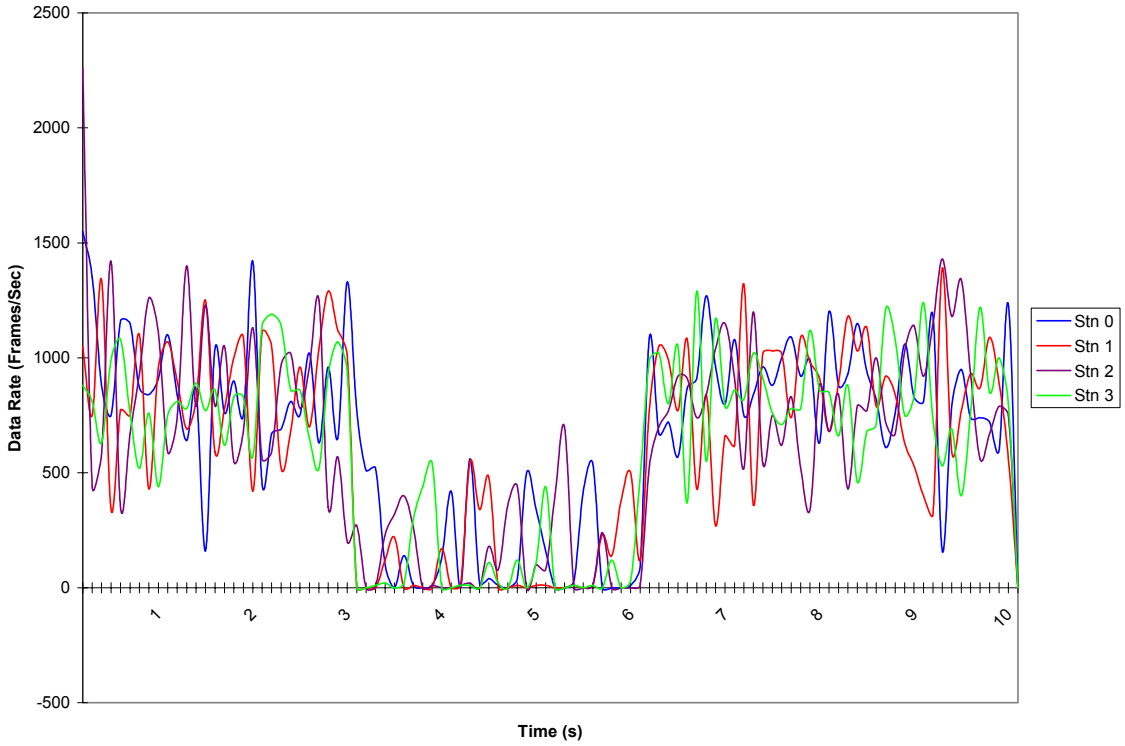

Figure 19. Simulation Results of RTS Attack Scenario (Attack Cycle 2000 frames/sec, Duration Field 310)

Simulation Time	Average Data Rate (Frames / sec)			
	Station 0	Station 1	Station 2	Station 3
0 - 3 sec	892.76	881.72	869.66	823.45
3 - 6 sec	**176.00**	**106.67**	**137.33**	**77.33**
6 - 10 sec	833.90	792.93	792.93	816.10

Table 7. Average Value from RTS Attack Scenario (Attack Cycle 2000 frames/sec, Duration Field 310)

Even though the attack cycle is much lower as compared with the deauthentication and disassociation cases, it was observed that all mobile stations suffered significant degradation in data rate. The degradation ranges from 80% (Station 0) to 91% (Station 3). Comparatively, the RTS is especially potent in denying network services to mobile stations.

Another simulation run was conducted with the same attack cycle, but the valued in duration field of the RTS frame was increased to 350. The results are illustrated in Figure 20 and Table 8.

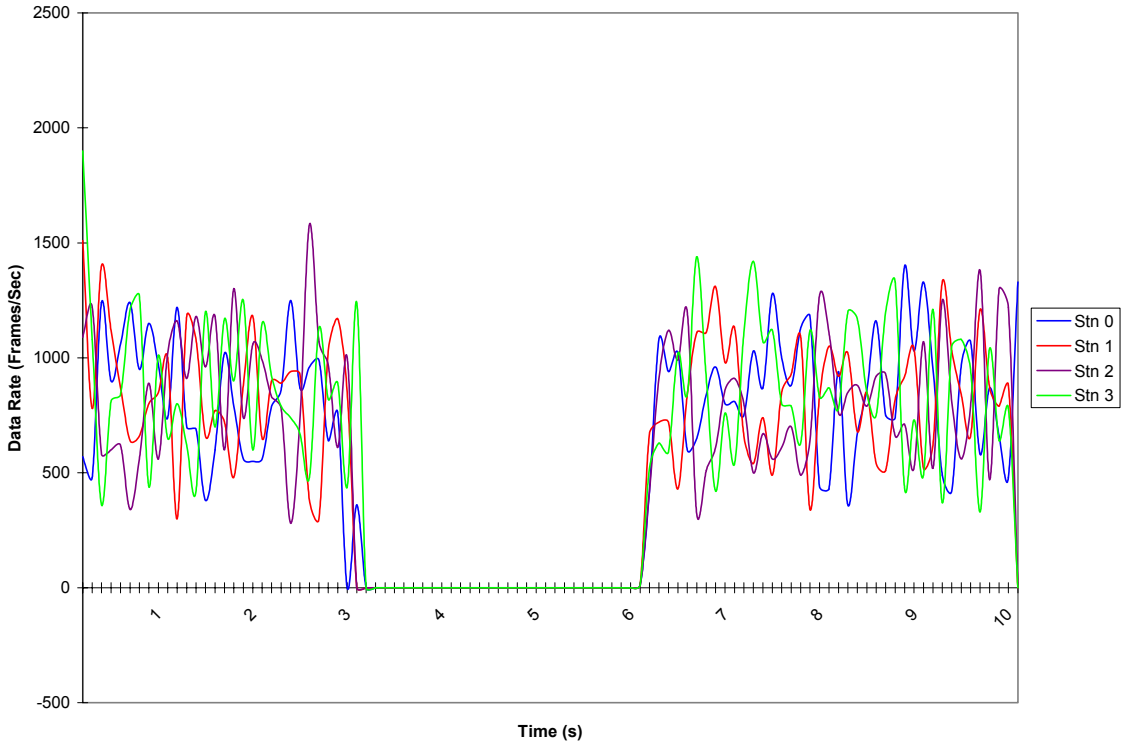

Figure 20. Simulation Results of RTS Attack Scenario (Attack Cycle 2000 frames/sec, Duration Field 350)

Simulation Time	Average Data Rate (Frames / sec)			
	Station 0	Station 1	Station 2	Station 3
0 - 3 sec	810.00	859.31	876.21	871.03
3 - 6 sec	**12.00**	**0.00**	**0.00**	**41.33**
6 - 10 sec	833.66	795.12	770.73	824.88

Table 8. Average Value from RTS Attack Scenario (Attack Cycle 2000 frames/sec, Duration Field 350)

Form the results in Table 20, it was observed that by only increasing the value in the duration field of the RTS frame, all mobile stations can be effectively denied of network services. All mobile stations will back off from accessing the medium in accordance with the value in the duration field of a CTS frame that was in response to the spoofed RTS frame. When the value in the duration field is sufficiently high with respect to the attack period; in this case the attack period was 500us, the attacker station can exclusively reserve the medium during the

time between each spoofed RTS frame, thus denying all other stations access to the medium.

To validate this conclusion, a simulation run was conducted with attack cycle set at 1000 frames/sec (Attack period 1000us) and duration field value set at 850. The results of the simulation are captured in Figure 21 and Table 9.

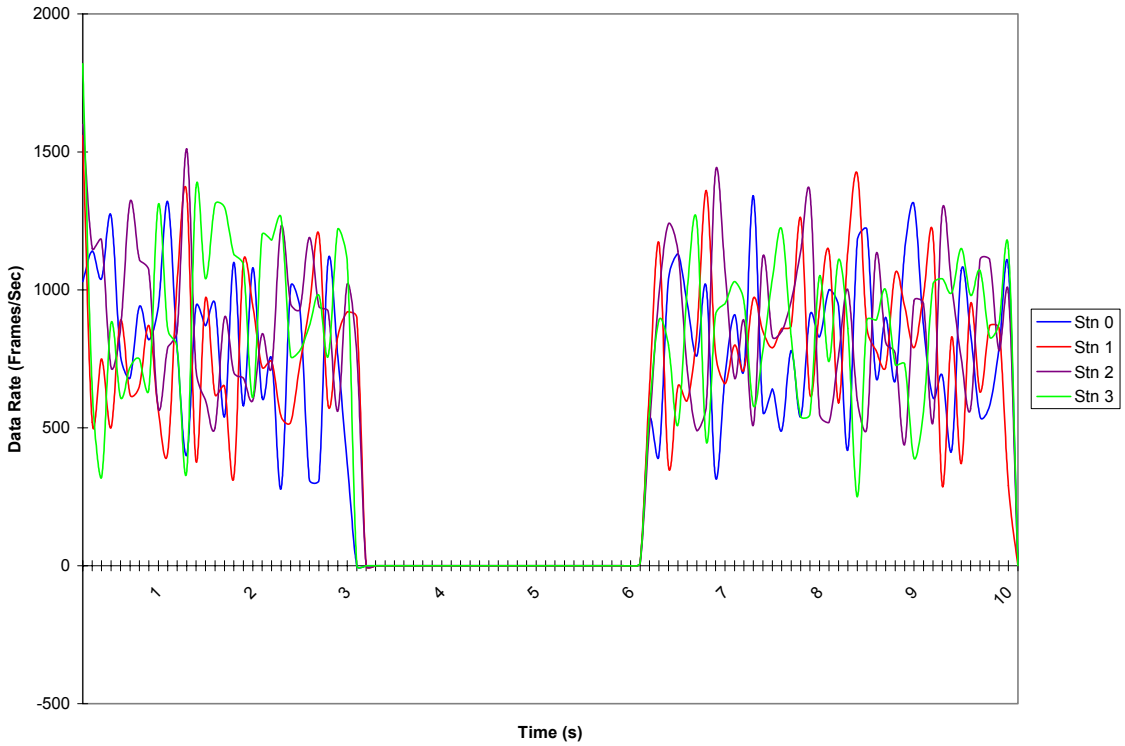

Figure 21. Simulation Results of RTS Attack Scenario (Attack Cycle 1000 frames/sec, Duration Field 850)

Simulation Time	Average Data Rate (Frames / sec)			
	Station 0	Station 1	Station 2	Station 3
0 - 3 sec	814.83	772.76	921.38	953.10
3 - 6 sec	**0.00**	**30.00**	**25.67**	**0.00**
6 - 10 sec	768.29	790.98	819.51	809.02

Table 9. Average Values from RTS Attack Scenario (Attack Cycle 1000 frames/sec, Duration Field 850)

The attack cycle has been reduced by 50% from 2000 to 1000 frames/sec and the value in the duration field increased to 850. Unlike the deauthentication

and disassociation attacks, the attack cycle of RTS is not as critical in determining the success of the attack. So long as the value of the duration field in the spoofed RTS frame was sufficiently high, the RTS attack would be able to deny all mobile stations access to the medium. The only drawback about the RTS attack is that the attacker station cannot choose to direct the attack at any single mobile station, but to attack all stations within the transmission range of the innocent collaborating station.

C. SIMULATION RESULTS OF DEFENSE SCENARIOS

The series of proposed defenses against the discussed attack are non-cryptographic countermeasures that can be implemented in the firmware of existing MAC hardware [3]. This has the advantages of hardening implemented IEEE 802.11-based WLANs, without having to make massive hardware replacements.

1. Defense Against Deauthentication and Disassociation Attacks

The deauthentication and disassociation vulnerability can be solved directly by explicitly authenticating IEEE 802.11 management frames, which are unencrypted at the moment. However, this measure will result in legacy MAC designs unable to meet the increased processing demands due to insufficient CPU capacity to implement this functionality as a software upgrade [3]. Therefore, solutions that are implemented at system-level with low overhead are preferred. In particular, by delaying the effects of deauthentication or disassociation frames, mobile stations has the opportunity to listen for subsequent frames from the AP before deciding to deauthenticate or disassociate from the AP. The mobile station can wait for a timeout period before obeying the deauthentication or disassociation frame. If before the timeout expires, the AP continues to communicate with the mobile station, the mobile station can disregard the received deauthentication or disassociation notification. This approach has the advantage that it can be implemented with a simple firmware modification to existing Network Interface Cards (NICs).

The mobile stations were enhanced with the timeout protection against spoofed deauthentication and disassociation frames. A simulation run was conducted with deauthentication attack cycle set at 4000 frames/sec and timeout value set at 500us. The results are illustrated in Figure 22 and Table 10.

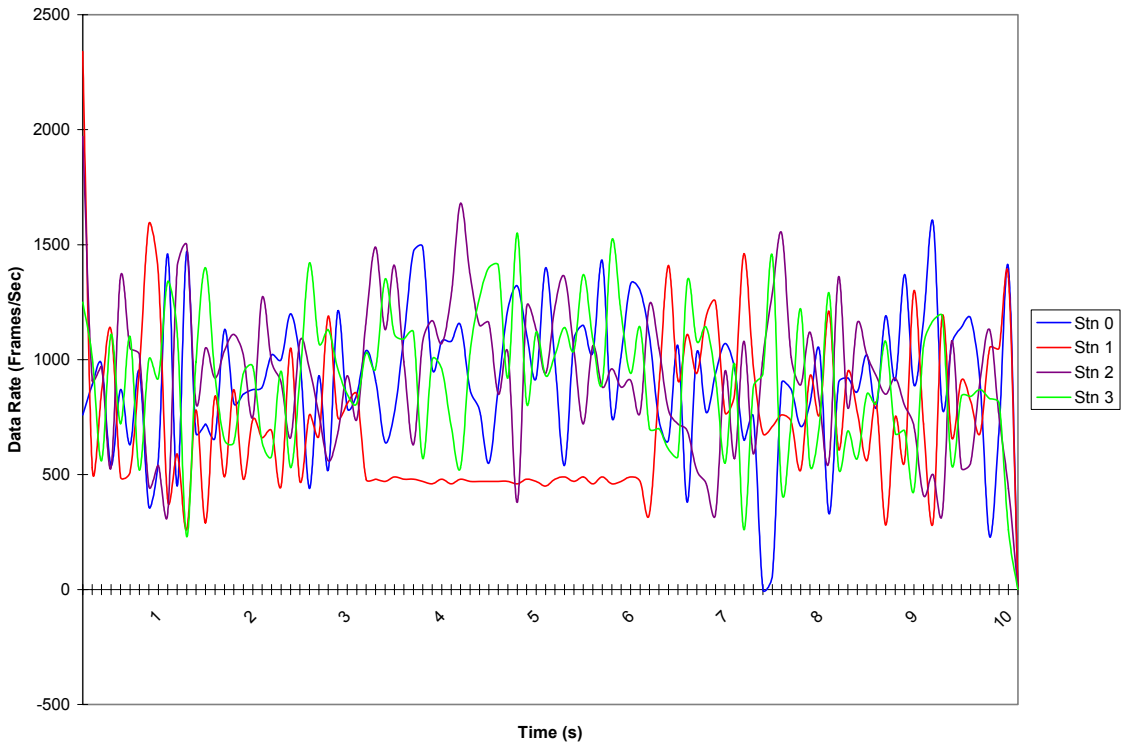

Figure 22. Simulation Results of Defense Against Deauthentication Scenario (Attack Cycle 4000 frames/sec, Timeout 500us)

Simulation Time	Average Data Rate (Frames / sec)			
	Station 0	Station 1	Station 2	Station 3
0 - 3 sec	849.66	793.45	948.97	912.07
3 - 6 sec	**1030.67**	**486.33**	**1070.67**	**1063.33**
6 - 10 sec	857.32	833.17	808.78	803.41

Table 10. Average Values From Defense Against Deauthentication Scenario (Attack Cycle 4000 frames/sec, Timeout 500us)

Comparing the results of this simulation run and that of the deauthentication attack at same attack cycle rate of 4000 frame/sec, but without defense enhancements (Figure 16, Table 4), it was observed that although the

hardened victim station suffers a data rate degradation of about 39%, this was still a significant improvement over the total denial of service in the unprotected situation.

It was also observed that unlike the unprotected situation, the other mobile stations in the network are not affected by the attack. This is because there was no longer a need for the victim station to aggressively reauthenticate with the AP, and thus other mobile stations' request to reserve the medium are not unnecessarily deferred due to the victim station's reauthenticate request frames.

The same defense enhancement was implemented against disassociation attacks. A simulation run was conducted with disassociation attacks at attack cycle of 5000 frames/sec and timeout at 500us. The results of the simulation are captured in Figure 23 and Table 11.

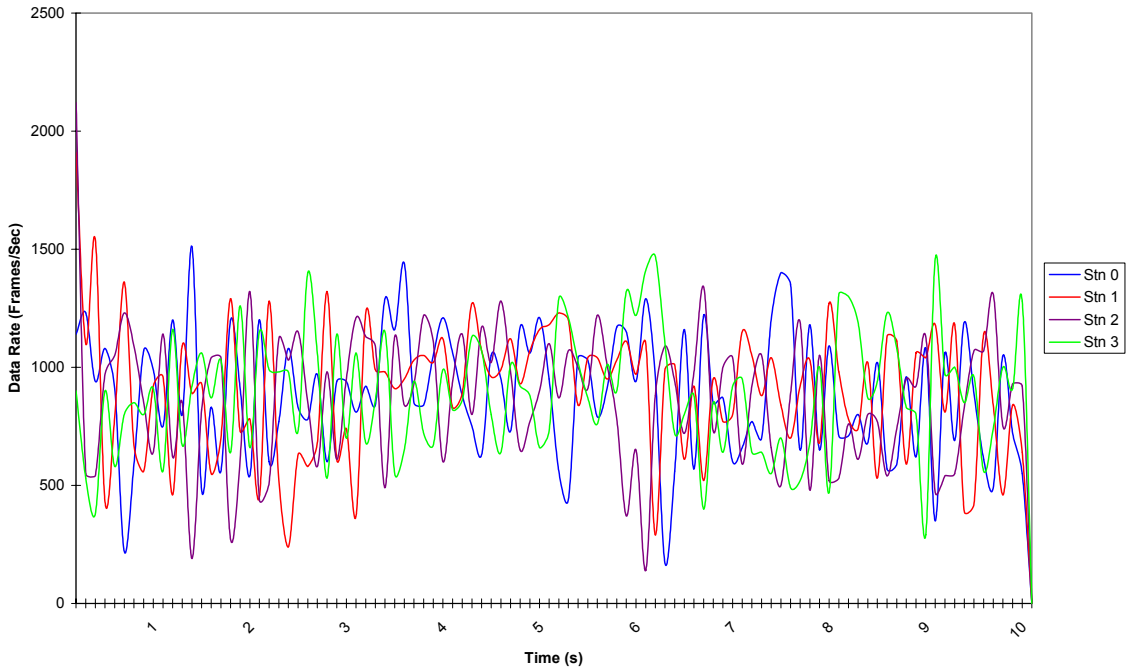

Figure 23. Simulation Results of Defense Against Disassociation Scenario (Attack Cycle 5000 frames/sec, Timeout 500us)

Simulation Time	Average Data Rate (Frames / sec)			
	Station 0	Station 1	Station 2	Station 3
0 - 3 sec	884.83	852.07	867.59	866.90
3 - 6 sec	962.67	1017.67	946.00	913.00
6 - 10 sec	810.73	839.02	802.44	863.41

Table 11. Average Values From Defense Against Disassociation Scenario (Attack Cycle 5000 frames/sec, Timeout 500us)

Comparing the results of this simulation run with those of the disassociation attack at attack cycle of 5000 frame/sec without defenses (Figure 18, Table 6), it was observed that the hardened victim station suffers no data rate degradation during the attack interval. The implemented defense was able to successfully defend against the disassociation attack at a rate that was previously successfully (97% data rate degradation).

Another simulation run was conduct with the attacker station intensifying the disassociation attack. The attack cycle is increased to 6667 frames/sec (attack period 150us). The results from this simulation are captured in Figure 24 and Table 12.

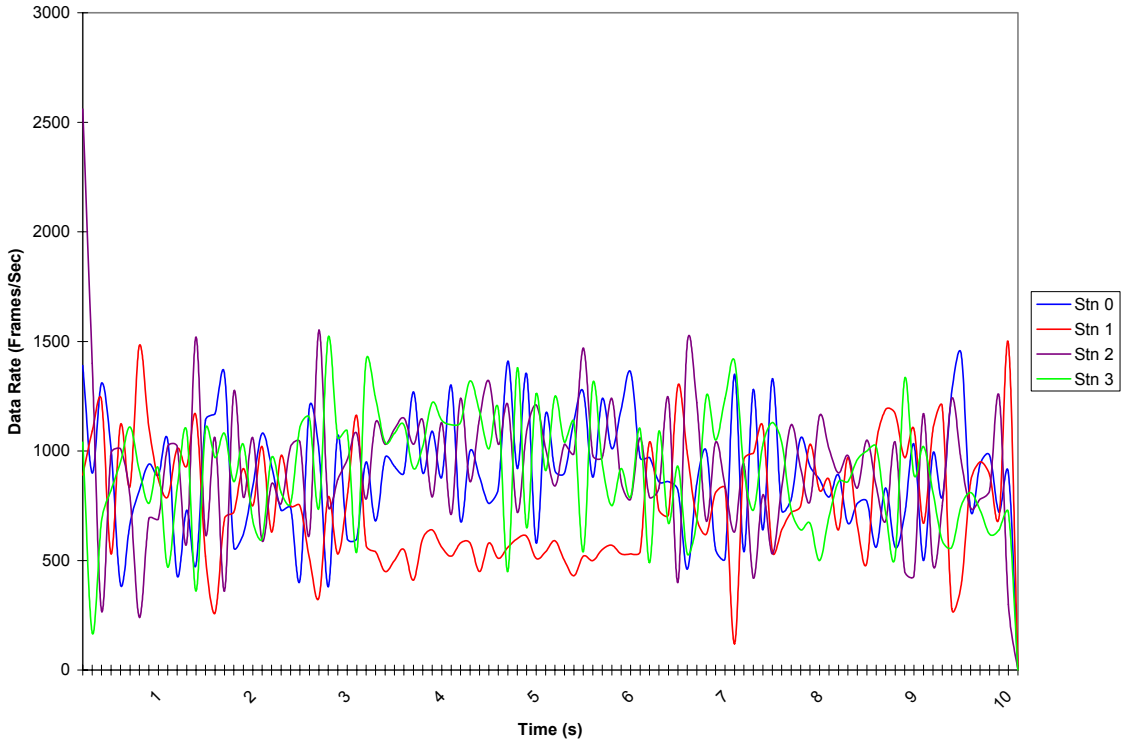

Figure 24. Simulation Results of Defense Against Disassociation Scenario (Attack Cycle 6667 frames/sec, Timeout 500us)

Simulation Time	Average Data Rate (Frames / sec)			
	Station 0	Station 1	Station 2	Station 3
0 - 3 sec	856.21	827.24	929.31	884.14
3 - 6 sec	998.33	**558.00**	1035.67	1033.33
6 - 10 sec	833.90	818.54	839.02	827.80

Table 12. Average Values From Defense Against Disassociation Scenario (Attack Cycle 6667 frames/sec, Timeout 500us)

The result of the simulation shows that the victim station suffers a data rate degradation of about 33%. This shows that the attacker station can still achieve limited success with the disassociation strategy, but needs to be very aggressive in mounting the attack in order to achieve the limited success. The increased in effort of conducting an attack and reduction in payoff may provide deterrence to an attacker from deploying this attack strategy.

2. Defense Against RTS Attacks

The virtual carrier-sense attack is much harder to defend against than the deauthentication, disassociation attack. This is mainly due to the problems of possible hidden nodes [5] in WLAN. Mobile stations may move in and out of an AP's coverage area, and thus may not be able to listen to all parts of a legitimate medium reservation handshake. These incomplete handshakes may seem like the RTS attacks of a malicious station, thus it is difficult to discern between legitimate reservation requests or malicious attacks.

One approach to mitigate the effects of an RTS attack is for mobile stations to monitor the state of communications in the WLAN. If there are deviations from the frame sequence or timing sequence as prescribe by the IEEE 802.11 protocol, mobile stations can take preventive measures to deny malicious attacks from succeeding in denial of network services. In the case of RTS attacks, mobile stations will continue to monitor the medium for a timeout period after a CTS frame has been received. The timeout period must take into consideration the expected response times of legitimate medium reservation and data transmission sequences (see Figure 7). If the expected response is not received after the timeout has expired, it could be due to the hidden node problem or the previous CTS frame could have been part of an attack on the network. Either case, the mobile station can ignore the previous medium reservation and commence to contend for medium access. If the previous medium reservation was legitimate, a collision will ensure, causing the mobile stations to backoff for a random period before trying again to reserve the medium for data transmission. If the previous medium reservation request was an attack conducted by a malicious station, the mobile stations can safely contend for the medium upon expiry of timeout, without the possibility of a collision

A simulation run was conducted with the attacker station conducting an RTS attack with attack cycle at 2000 frames/sec and duration field value at 350. The mobile stations are implemented with the described defenses, with the timeout value set at 1 DIFS (128us). The results of the simulation are captured in Figure 25 and Table 13.

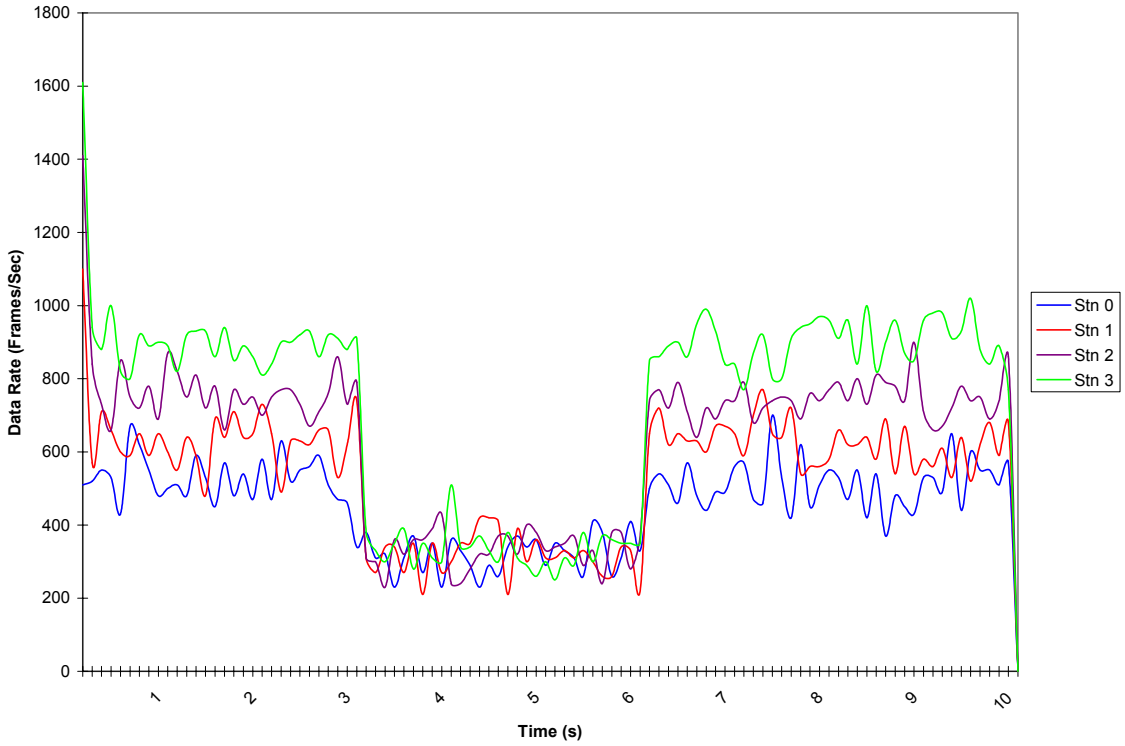

Figure 25. Simulation Results of Defense Against RTS Attack Scenario (Attack Cycle 2000 frames/sec, Duration Field 350, Timeout 128us)

Simulation Time	Average Data Rate (Frames / sec)			
	Station 0	Station 1	Station 2	Station 3
0 - 3 sec	528.28	638.97	777.24	914.48
3 - 6 sec	**319.67**	**334.67**	**346.00**	**353.00**
6 - 10 sec	495.37	599.76	717.32	863.90

Table 13. Average Values From Defense Against RTS Attack Scenario (Attack Cycle 2000 frames/sec, Duration Field 350, Timeout 128us)

It was observed from the results of the simulation that the defense enhancements made to the mobile stations achieved limited success in mitigating the effects of an RTS attacks. The mobile stations suffer from degradation to data rates ranging from 39% (Station 1) to 61% (Station 3). The degradation in the data rates is attribute to the excess wait time (1 DIFS) necessary to determine that the medium is free, before commencing to contend for medium access. Comparing with the almost total denial of network services

for an unprotected system (Figure 20, Table 8), the degradation suffered by the protected system may still be acceptable for users of the WLAN.

Another simulation was conducted with the attacker station set up with attack cycle of 1000 frames/sec and duration field value set at 850. The mobile stations' timeout value remained at 1 DIFS. The results of this simulation are illustrated in Figure 26 and Table 14.

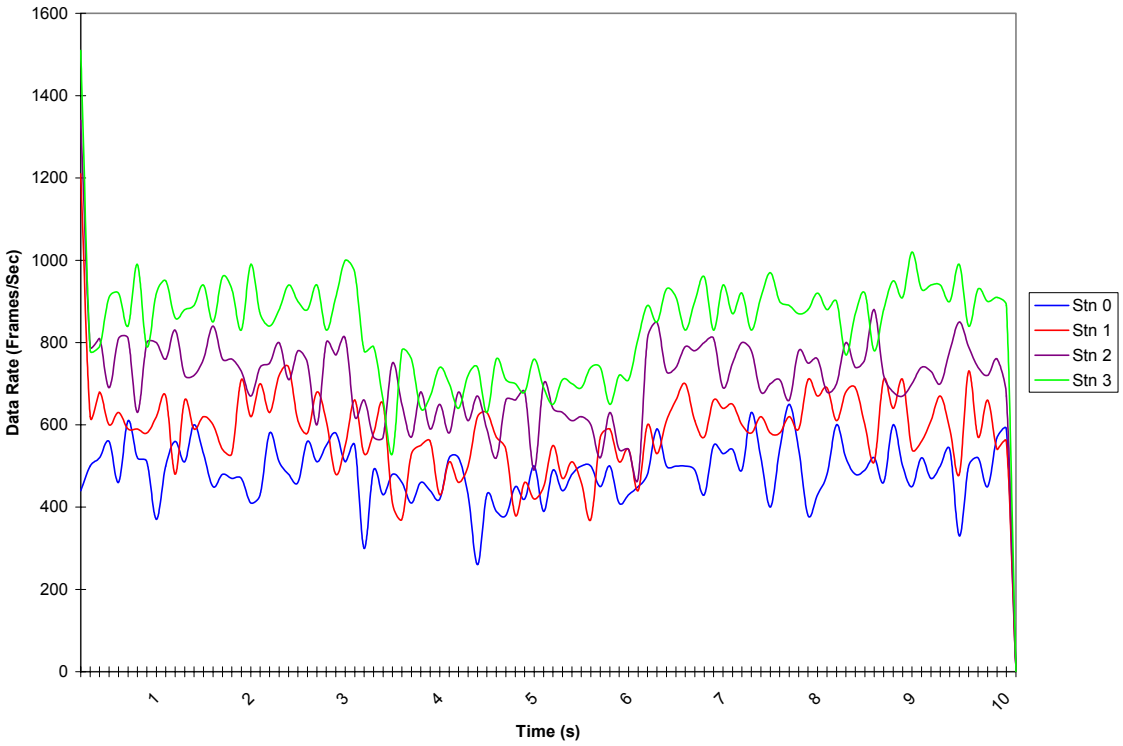

Figure 26. Simulation Results of Defense Against RTS Attack Scenario (Attack Cycle 1000 frames/sec, Duration Field 850, Timeout 128us)

Simulation Time	Average Data Rate (Frames / sec)			
	Station 0	Station 1	Station 2	Station 3
0 - 3 sec	504.83	635.86	777.24	914.48
3 - 6 sec	444.33	512.67	616.00	712.00
6 - 10 sec	493.41	599.27	723.17	874.63

Table 14. Average Values From Defense Against RTS Attack Scenario (Attack Cycle 1000 frames/sec, Duration Field 350, Timeout 128us)

Analyzing the results from the simulation, it was expected that the implement defense against RTS attacks performed better when the attack is less aggressive; 50% reduction in attack cycle rate. The data rate degradation suffered by the mobile station ranges from 12% to 22%, as compared to the previous simulation case of 39% to 61%, and almost total denial of service in the unprotected system. As such, the implemented defense can be an effective deterrent to potential attackers as the attack effort must be aggressive enough to achieve the desired level of data rate degradation to mobile stations.

However, more sophisticated attackers can outsmart this defense enhancement by simply mimicking as two legitimate stations (1 AP and 1 mobile station) exchanging medium access handshake and data transmissions. This will deceive the legitimate mobile station into backing off from accessing the medium, until such a time the attacker station decides to stop the spoofed exchange.

D. SUMMARY OF EFFECTIVENESS OF IMPLEMENTED DEFENSES

It was observed that different attack genres have different effects on the data rate of the victim mobile station. The results from the series of simulation runs are tabulated in Table 15.

Attack Genre		Avg Data Rate Degradation	
	Attack Cycle Rate (frames/sec)	Unprotected Systems	Hardened Systems
Deauthentication Attacks	4000	100%	39%
Disassociation Attacks	5000	97%	0%
RTS Attacks	2000	100%	50%

Table 15. Consolidated Data Rate Degradation

From the table, it was observed that the hardened systems were able to resist all three kinds of attacks to different degrees, as compared to the

unprotected systems, where there was almost total denial of service to the victim stations.

Although the hardened mobile systems were able to resist the attacks, it was also shown in the simulation runs that by increasing the aggressiveness of the attacks, the attacker could still achieve success in limiting the victim station's data rate. However, it is hope that the higher demands placed on the attacker to mount the attack on hardened mobile stations, coupled with the reduced payoff of the attack, will discourage an attacker from deploying the 3 genres of attacks.

The series of proposed defenses against the discussed attack were conceived such that they can be implemented in the firmware of existing MAC hardware. This has the advantages of hardening implemented IEEE 802.11-based WLANs, without having to make massive hardware replacements. The benefits of the ability to limit the success of attacks and the need for only firmware upgrades to implement the proposed solutions are worthwhile for the proposed solutions to be considered, not as a permanent solution, but as interim measures taken to discourage attackers from exploiting the known weaknesses. The long-term objective must still be to eliminate implicit trust IEEE 802.11 networks place in a transmitting station's address, and to implement appropriate per-packet authentication mechanisms such that trust among the nodes of a WLAN is explicitly established and not implicitly accepted.

VI. CONCLUSION

WLANs based on the IEEE 802.11 protocol have been widely deployed in many areas. This is mainly due to the physical conveniences of wireless mobile communications. However, there are some vulnerabilities that exist in the IEEE 802.11 protocol that make the implement WLANs susceptible to attacks that target the confidentiality of data and denial of network services. This work focused on those vulnerabilities that can be exploited to mount DoS attacks on specific victim stations or to conduct network wide DoS attacks.

The two categories of vulnerabilities of the IEEE 802.11 protocol; identity vulnerabilities and medium access vulnerabilities can be exploited to conduct attacks that deauthenticate or disassociate mobile station from the network, or to indefinitely reserve the medium such that legitimate mobile stations cannot access the medium. These attack strategies have the common objective of denying the victim mobile stations access to the network.

In response to the 3 genres of attacks, specific countermeasures are discussed and simulated to evaluate their effectiveness against the attacks. The results from the simulation runs demonstrated that these countermeasures do indeed result in increased resistance against the attacks. The benefits of the proposed countermeasures are that they can be implemented only with firmware upgrades, and thus eliminating the need for massive replacement of existing network hardware.

However, the fundamental problem of implicit trust IEEE 802.11 networks place in a transmitting station’s address remains. Therefore the long-term plan must be to find solutions to eradicate the implicit trust problem.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. SOURCE CODE LIST OF SIMULATION MODEL

```
//--------------------------------------------------------------
// File: Dyna.ned
//
//  OMNeT++ simulation of IEEE802 network
//
//  Nicholas Tan, 01 September 2003
//  US Naval Postgraduate School
//--------------------------------------------------------------

//--------------------------------------------------------------


// Access Point --
//
// An Access Point (AP) of the Wireless LAN
//
simple AccessPoint //
   parameters:
   gates:
      out: out;
      in: in;
endsimple


// Freespace --
//
// The wireless transmission medium of the network
//
simple Freespace //
   parameters:
   gates:
      out: out[];
      in: in[];
      out: APout;
      in: APin;
      out: Aout;
      in: Ain;
endsimple


// Station --
//
// the Mobile station (STA) of the wireless network
//
simple Station //
   gates:
      out: out;
      in: in;
      in: Cin;
endsimple


//Hacker attacking the network
simple Attacker //
   gates:
      in: in;
      out: out;
endsimple

//Wireless Network--
//
// Model of the IEEE802 wireless network, consisting of serveral mobile stations, an access point
//
module IEEE802 //
   parameters:
```

```
        num_station : numeric const;
    submodules:
        ap: AccessPoint; //
            parameters:
            display: "p=164,110;i=access_point02;b=80,75";
        freespace: Freespace; //
            parameters:
            gatesizes:
                in[num_station],
                out[num_station];
            display: "p=245,199;i=cloud_l;b=103,49";
        sta: Station[num_station]; //
            parameters:
            display: "p=106,317,r,50;i=pda1;b=25,36";
        attacker: Attacker;
            display: "p=440,150;i=hacker05;b=139,105";
        clk: Clock[num_station];
            display: "p=78,340,r,50;i=cogwheel2_s;b=15,10";
    connections:
        for i=0..num_station-1 do
            sta[i].out --> datarate 11000000 --> freespace.in[i]; //
            freespace.out[i] --> sta[i].in; //
            clk[i].out --> sta[i].Cin;
        endfor;
        ap.out --> datarate 11000000 --> freespace.APin; //
        freespace.APout --> ap.in; //
        freespace.Aout --> attacker.in display "m=m,95,63,47,28"; //
        attacker.out --> datarate 11000000 --> freespace.Ain display "m=m,47,28,95,63"; //

    display: "p=2,4;b=638,464";
endmodule


simple Clock
    gates:

        out: out;
endsimple
//Wireless LAN
//
// Instantiates an IEEE 802 wireless network.
//
network theWirelessLAN : IEEE802
    parameters:
        num_station = input(8,"Number of Mobile Station :");
endnetwork
```

```
//---------------------------------------------------------------
// file: ap.ccp
//(part of WirelessLAN - an OMNeT++ simulation of IEEE802)
//---------------------------------------------------------------

#include "omnetpp.h"

class AccessPoint : public cSimpleModule
{
   Module_Class_Members(AccessPoint,cSimpleModule,16384)
   virtual void activity();
};

Define_Module( AccessPoint );

void AccessPoint::activity()
{
   //recording variables
            cOutVector resp_v("DataStn");


            float process_time = 0;
            float time_stamp = 0;
            float elasped_time = 0;
            float delay = 0;

   int random_wait; // variance time randomised

            //Address Information
            int sta_add = parentModule()->par("num_station");
            int AP_addr = sta_add + 1;
            const int data_addr = 99; //permanent addr for data sta

  //msg parameters
  int source; // msg source
  int destination; // msg destination
  int msg_kind; //msg kind
  long msg_nav; // nav of the msg
  float msg_timestamp; // time stamp of msg



  //IEEE802 vectors
  //-------------------------
  //State definitions
  int State = 1;
  //1 - Unauthenticated, Unassociated
  //2 - Authenticated, Unassociated
  //3 - Authenticated, Associated

  //Network Allocation Vector (NAV) - max value 32767 (appox 32 msec)

  long NAV = 0;
  long temp_NAV = 0; // updated NAV value of the incoming packet

  //Flags

  bool flag_AUT = false; // is there an outstanding AUT request
  bool flag_ASS = false; // is there an outstanding ASS request
  bool flag_CTS = false; // This station has CTS
  bool flag_RTS = false; // This station has RTS outstanding
  bool flag_ORTS = false;// Another Station has RTS outstanding
  bool flag_OCTS = false;// Another Station has CTS outstanding
  bool flag_CLK = false; // not waiting for CLK signal



  //Timing Constants - dependant on the PHY
                  //Device Interval - delays due to busy device
```

```
                    const int Device_Itvl = 3; // 3 cycles of clock

                    //DIFS
                    const int DIFS = 128; // microsecond
                    const int error_DIFS = 12; // microsecond

                    //SIFS
                    const int SIFS = 28; // microsecond
                    const int error_SIFS = 4; // microsecond

                    //Clock Cycle Time
                    const int CLOCK = 4; // microsecond

                    //microsecond denominator
                    const double Million = 0.000001;


          // Message Kind Definition
          const int RTS = 1;
          const int CTS = 2;
          const int AUT = 3;
          const int ASS = 4;
          const int rply_AUT = 5;
          const int rply_ASS = 6;
          const int de_AUT = 7;
          const int de_ASS = 8;
          const int DATA = 9;
          const int ACK = 10;
          const int CLK = 11;
          const int LASTPKT = 12;

//---------------------------------------------------------------------------


for(;;)

{//for

          //receiving from FS
          cMessage *done = receive();
          source = done->scr_add();
          destination = done->des_add();
          msg_kind = done->kind();
          msg_nav = done->nvec();
          msg_timestamp = done->msgtimer();

          delete done;


if (destination == AP_addr)
{//if destination

          if (msg_kind == AUT)
          {//if AUT

               //reply with a rply_AUT
                         cMessage *work = new cMessage( "rply_AUT" );
                         work->setSCR(AP_addr);
                         work->setDES(source);
                         //work->setROUTESCR(AP_addr);
                         //work->setROUTDES(source);
                         work->setKind(rply_AUT);
                         work->setMsgTime(simTime());
                         work->setNAV(0);

                         random_wait = intrand(6);
                         wait ((SIFS+random_wait)*Million);

                         send( work, "out" );
```

```
}//if AUT


if(msg_kind == ASS)
{//if ASS

                        //reply with a rply_ASS
                        cMessage *work = new cMessage( "rply_ASS" );
                        work->setSCR(AP_addr);
                        work->setDES(source);
                        //work->setROUTESCR(AP_addr);
                        //work->setROUTDES(source);
                        work->setKind(rply_ASS);
                        work->setNAV(0);
                        work->setMsgTime(simTime());

                        random_wait = intrand(6);
                        wait ((SIFS+random_wait)*Million);

                        send( work, "out" );

}//if ASS


if((msg_kind == RTS))
{//if RTS reply with CTS

                        //reply with a CTS
                        cMessage *work = new cMessage( "CTS" );
                        work->setSCR(AP_addr);
                        work->setDES(source);
                        //work->setROUTESCR(AP_addr);
                        //work->setROUTDES(source);
                        work->setKind(CTS);

                        work->setMsgTime(simTime());

                        // The transmission delay from RTS to CTS needs to be subtracted
                        // from the RTS NAV

                        process_time = msg_nav - SIFS;

                        work->setNAV(process_time);

                        NAV = process_time; //setting own NAV vector
                        time_stamp = simTime(); //of the NAV vector

                        wait ((SIFS)*Million);

                        send( work, "out" );


}//if RTS


if(msg_kind == LASTPKT)
{//if DATA

                        //reply with a ACK for last packet
                        cMessage *work = new cMessage( "ACK" );
                        work->setSCR(AP_addr);
                        work->setDES(source);
                        //work->setROUTESCR(AP_addr);
                        //work->setROUTDES(source);
                        work->setKind(ACK);
                        work->setNAV(0);

                        wait ((SIFS)*Million);//delay SIFS
                        work->setTimestamp();
```

```
                                send( work, "out" );
                                //resp_v.record(source); //xxx

            }//if DATA

}//if destination


} //for
}//void
```

```
//-------------------------------------------------------------
// file: clock.ccp
//(part of WirelessLAN - an OMNeT++ simulation of IEEE802)
//-------------------------------------------------------------

#include "omnetpp.h"

class Clock : public cSimpleModule
{
   Module_Class_Members(Clock,cSimpleModule,16384)
   virtual void activity();
};

Define_Module( Clock );

void Clock::activity()
{


        //Addressing Information
        int sta_add = parentModule()->par("num_station"); //total num of station
        int AP_addr = sta_add + 1; //permanent AP address
        int Attk_addr = sta_add + 2; // permanent addr for attacker
        int Clk_addr = sta_add + 3; // permanet addr for clock
        const int data_addr = 99; //permanent addr for data sta



  //IEEE802 vectors
  //--------------------------
  //State definitions
  int State = 1;
  //1 - Unauthenticated, Unassociated
  //2 - Authenticated, Unassociated
  //3 - Authenticated, Associated

  //Network Allocation Vector (NAV) - max value 32767 (appox 32 msec)

  long NAV = 0;
  long temp_NAV = 0; // updated NAV value of the incoming packet

  //Flags

  bool flag_AUT = false; // is there an outstanding AUT request
  bool flag_ASS = false; // is there an outstanding ASS request
  bool flag_CTS = false; // This station has CTS
  bool flag_RTS = false; // This station has RTS outstanding
  bool flag_ORTS = false;// Another Station has RTS outstanding
  bool flag_OCTS = false;// Another Station has CTS outstanding
  bool flag_CLK = false; // not waiting for CLK signal



  //Timing Constants - dependant on the PHY
                //Device Interval - delays due to busy device
                const int Device_Itvl = 3; // 3 cycles of clock

                //DIFS
                const int DIFS = 128; // microsecond
                const int error_DIFS = 12; // microsecond

                //SIFS
                const int SIFS = 28; // microsecond
                const int error_SIFS = 4; // microsecond

                //Clock Cycle Time
                const int CLOCK = 4; // microsecond

                //microsecond denominator
                const double Million = 0.000001;
```

```
        // Message Kind Definition
        const int RTS = 1;
        const int CTS = 2;
        const int AUT = 3;
        const int ASS = 4;
        const int rply_AUT = 5;
        const int rply_ASS = 6;
        const int de_AUT = 7;
        const int de_ASS = 8;
        const int DATA = 9;
        const int ACK = 10;
        const int CLK = 11;
        const int LASTPKT = 12;

//-------------------------------------------------------------------------

double delay = CLOCK * Million;


  for(;;)
   {

                        cMessage *work = new cMessage( "clock" );
                        work->setSCR(Clk_addr);
                        work->setDES(Clk_addr);
                        work->setKind(CLK);//clock signal
                        work->setNAV(0);
                        wait (delay);
                        send( work, "out" );

        } //for
}//void
```

```
//--------------------------------------------------------------
// file: freespace.ccp
// (part of Wireless LAN - an OMNeT++ simulation of IEEE802)
//--------------------------------------------------------------

#include "omnetpp.h"

class Freespace : public cSimpleModule
{
   Module_Class_Members(Freespace,cSimpleModule,16384)
   virtual void activity();
};

Define_Module( Freespace );

void Freespace::activity()
{

  // address of all stations
  int sta_add = parentModule()->par("num_station"); // Total num of STA
  int AP_addr = sta_add + 1; // permanent addr for AP
  int Attk_addr = sta_add + 2; // permanent addr for attacker


  // msg parameters
  int source; // source MAC address of the rx msg
  int msg_kind; //type of msg
  long msg_nav; // nav of msg
  float msg_timestamp; //timestamp of msg
  //float process_time;



  int loop_count;

   //IEEE802 vectors
   //-------------------------
   //State definitions
   int State = 1;
   //1 - Unauthenticated, Unassociated
   //2 - Authenticated, Unassociated
   //3 - Authenticated, Associated

   //Network Allocation Vector (NAV) - max value 32767 (appox 32 msec)

   long NAV = 0;
   long temp_NAV = 0; // updated NAV value of the incoming packet

   //Flags

   bool flag_AUT = false; // is there an outstanding AUT request
   bool flag_ASS = false; // is there an outstanding ASS request
   bool flag_CTS = false; // This station has CTS
   bool flag_RTS = false; // This station has RTS outstanding
   bool flag_ORTS = false;// Another Station has RTS outstanding
   bool flag_OCTS = false;// Another Station has CTS outstanding
   bool flag_CLK = false; // not waiting for CLK signal



   //Timing Constants - dependant on the PHY
                        //Device Interval - delays due to busy device
                        const int Device_Itvl = 3; // 3 cycles of clock

                        //DIFS
                        const int DIFS = 128; // microsecond
                        const int error_DIFS = 12; // microsecond

                        //SIFS
                        const int SIFS = 28; // microsecond
```

```
                const int error_SIFS = 4; // microsecond

                //Clock Cycle Time
                const int CLOCK = 4; // microsecond

                //microsecond denominator
                const double Million = 0.000001;


        // Message Kind Definition
        const int RTS = 1;
        const int CTS = 2;
        const int AUT = 3;
        const int ASS = 4;
        const int rply_AUT = 5;
        const int rply_ASS = 6;
        const int de_AUT = 7;
        const int de_ASS = 8;
        const int DATA = 9;
        const int ACK = 10;
        const int CLK = 11;
        const int LASTPKT = 12;

//---------------------------------------------------------------------------


for(;;)
   {


                //receiving one packet
                cMessage *pkt = receive();

                source = pkt->scr_add();
                msg_kind = pkt->kind();
                msg_nav = pkt->nvec();
                msg_timestamp = pkt->msgtimer();



                // rebro packet

                if (source > sta_add) // either AP or Attacker packet

                { // AP or Attk

                        //broadcast to all STA
                        for(loop_count=0; loop_count < sta_add; loop_count++)

                        {//for
                                int dest = loop_count;
                                cMessage *work = (cMessage *) pkt->dup(); //duplicate orginal pkt
                                work->setNAV(msg_nav); //omnet bug, need to explicite include this
                                work->setMsgTime(msg_timestamp);
                    send( work, "out", dest);

                        }//for

                        if (source == AP_addr)

                        {//AP packet
                        // send to Attk
                   cMessage *work3 = new cMessage( *pkt ); //duplicate orginal pkt
                        work3->setNAV(msg_nav); //omnet bug, need to explicite include this
                        work3->setMsgTime(msg_timestamp);
                        send( work3, "Aout" );

                        }// AP packet
```

```
                    else

                    {//Attk packet
                    // send to AP
                cMessage *work2 = new cMessage( *pkt ); //duplicate orginal pkt
                    work2->setNAV(msg_nav); //omnet bug, need to explicite include this
                    work2->setMsgTime(msg_timestamp);
                    send( work2, "APout" );

                    }
                } // AP or Attk

                else

                {//else

                    // send to AP
                cMessage *work4 = (cMessage *) pkt->dup(); //duplicate orginal pkt
                    work4->setNAV(msg_nav); //omnet bug, need to explicite include this
                    work4->setMsgTime(msg_timestamp);
                    send( work4, "APout" );


                    // send to Attaker
                cMessage *work5 = new cMessage( *pkt ); //duplicate orginal pkt
                    work5->setNAV(msg_nav); //omnet bug, need to explicite include this
                    work5->setMsgTime(msg_timestamp);
                    send( work5, "Aout" );

                    // broadcast to all other STA

                    for(loop_count=0; loop_count < sta_add; loop_count++)

                    {//for
                        int dest = loop_count;
                        if (loop_count != source)
                        {//if
                        cMessage *work = new cMessage( *pkt );
                        work->setNAV(msg_nav); //omnet bug, need to explicite include this
                        work->setMsgTime(msg_timestamp);
                        send( work, "out", dest);

                        }//if
                    }//for


                }//else

        delete pkt;


        } //for

}//void
```

```
//---------------------------------------------------------------
// file: sta(unprotected).ccp
//       (part of WirelessLAN - an OMNeT++ simulation of IEEE802)
//---------------------------------------------------------------

#include "omnetpp.h"


class Station : public cSimpleModule
{
   Module_Class_Members(Station,cSimpleModule,16384)
   virtual void activity();
};

Define_Module( Station );

void Station::activity()
{
        float     process_time = 0;   // working variable for internal calculations
        float     time_stamp = 0;             // time stamp of NAV
        float     elasped_time = 0;   // working variable for internal calculations
        float     tx_time = 0;                // Time between successive packet transmissions
        float     tx_stamp = 0;               // Time stamp of packet sent, used in conjunction with
tx_time
        int             delay_len = 0;              // working variable for internal calculations
        int             delay_counter = 0;  // working variable for internal calculations
        int             true_counter = 0;
        int             packet_count = 0;
        int             delay_RTS;                  // random back off SIFS for RTS frame
        int             delay_CTS;                  // delay for CTS tx
        int             data_len = 120;             // arbitual data transmission time
        time_t    current_time;               // stores current pc clock time in seconds
        ldiv_t    divresult;                  // used for long division of integers in random num
generation


        //recording variables
        cOutVector resp_v("DataStation");       // time when data is transmitted
        cOutVector resp_b("Tx times");                  // time between successive transmissions

        //Addressing Information
        int             own_addr = gate( "out" )->toGate()->index();        //sta own address
        int             sta_add = parentModule()->par("num_station");     //total num of station
        int             AP_addr = sta_add + 1;
//permanent AP address
        int             Attk_addr = sta_add + 2;
//permanent addr for attacker
        const int data_addr = 99;
//permanent addr for data sta

        //data parameters
        const int data_length = 8000; //1000 bytes (xxx Need to check on this one)

  //msg parameters
  int           source;                     // msg source
  int           destination;          // msg destination
  int           msg_kind;                   //msg kind
  long          msg_nav;              // nav of the msg
  float     msg_timestamp;     // time stamp of msg


  //IEEE802 vectors
  //--------------------------
  //State definitions
  int State = 1; //initial state
  //1 - Unauthenticated, Unassociated
  //2 - Authenticated, Unassociated
  //3 - Authenticated, Associated
```

```
  //Network Allocation Vector (NAV) - max value 32767 (appox 32 msec)

  long NAV = 0;
  long temp_NAV = 0; // updated NAV value of the incoming packet

  //Flags

  bool flag_AUT = false;        // is there an outstanding AUT request
  bool flag_ASS = false;        // is there an outstanding ASS request
  bool flag_CTS = false;        // This station has CTS
  bool flag_RTS = false;        // This station has RTS outstanding
  bool flag_ORTS = false;       // Another Station has RTS outstanding
  bool flag_OCTS = false;       // Another Station has CTS outstanding
  bool flag_CLK = false;        // not waiting for CLK signal
  bool flag_DataSent;                   // Data flag to indicate if there is a need to send the last packet


  //Timing Constants - dependant on the PHY
                        //Device Interval - delays due to busy device
                        int Device_Itvl; // device delay

                        //DIFS
                        const int DIFS = 128;                   // microsecond
                        const int error_DIFS = 12;      // microsecond

                        //SIFS
                        const int SIFS = 28;            // microsecond
                        const int error_SIFS = 4;       // microsecond

                        //Clock Cycle Time
                        const int CLOCK = 4;                    // microsecond

                        //microsecond denominator
                        const double Million = 0.000001;


              // Message Kind Definition
              const int RTS = 1;
              const int CTS = 2;
              const int AUT = 3;
              const int ASS = 4;
              const int rply_AUT = 5;
              const int rply_ASS = 6;
              const int de_AUT = 7;
              const int de_ASS = 8;
              const int DATA = 9;
              const int ACK = 10;
              const int CLK = 11;
              const int LASTPKT = 12;

//------------------------------------------------------------------------

for(;;)
{//for

//randomly generate device interval for each cycle
current_time = time(NULL);
srand(current_time+own_addr);
divresult = ldiv (current_time,31);//the max possible RNG for intuniform is 31
Device_Itvl = intuniform(4, 7, divresult.rem);

//-----------------Incoming Pkt ------------------------


//receiving from FS
cMessage *done = receive();
source = done->scr_add();
destination = done->des_add();
```

```
msg_kind = done->kind();
msg_nav = done->nvec();
msg_timestamp = done->msgtimer();
delete done;


//-----------------------CLOCK Signal----------------------------
if (msg_kind == CLK)
{//if clock signal

        if (NAV > 0)
        {//if NAV >0
        NAV = NAV - CLOCK;
        if (NAV <=0) NAV = 0;
        }//if NAV >0



        time_stamp = simTime();

}//if clock signal



//---------------PACKETs addressed to this station----------------
if (destination == own_addr)
        {// if destination


//------------------Authentication & Association------------------
        if ((msg_kind == rply_AUT) && (flag_AUT)) {State =2;flag_AUT = false; flag_ASS = false;}
        if ((msg_kind == rply_ASS) && (flag_ASS)) {State =3;flag_ASS = false;}
        if (msg_kind == de_AUT) {State =1; flag_AUT = false;flag_ASS = false;}
        if ((msg_kind == de_ASS) && (State ==3)) State =2;


//--------------------------CTS-----------------------------------

        if ((msg_kind == CTS) && (State == 3) && (flag_RTS))
        {//if CTS

                //----NAV Calculuation---------------
                NAV = msg_nav; //set the NAV to that specified in CTS

                flag_CTS = true; //CTS has been approved for this station
                flag_DataSent = false; //no need to send last packet, only need if the first packets are sent


                //delay 1 SIFS and listen out for other Transmitions
                //if there is any transmission during the SIFS
                //the CTS will be aborted (flag_CTS set to false)
                flag_CLK = false;
                delay_counter = 1;
                delay_CTS = (SIFS / CLOCK)  + intrand(error_SIFS/CLOCK); //DIFS + x number of clock
cycle

                while (delay_counter <= delay_CTS)// SIFS + x number of clock cycle
                        {//receiving from CLK
                        cMessage *done = receive();
                        destination = done->des_add();
                        msg_kind = done->kind();
                        msg_nav = done->nvec();
                        delete done;
                        if (msg_kind != CLK)
                                {//if
                                if (msg_kind == RTS) {flag_CTS = false; flag_RTS = false;NAV =
msg_nav; delay_counter = (delay_CTS + 10); if (NAV <= 0) NAV = 0;}//back off because someone else is transmitting
RTS
```

```
                                        if (msg_kind == CTS) {flag_CTS = false; flag_RTS = false;NAV =
msg_nav; delay_counter = (delay_CTS + 10); if (NAV <= 0) NAV = 0;}//back off because someone else is transmitting
CTS
                                        if (msg_kind == DATA) {flag_CTS = false; flag_RTS = false;NAV =
msg_nav; delay_counter = (delay_CTS + 10); if (NAV <= 0) NAV = 0;}//back off because someone else is transmitting
DATA
                                        if ((msg_kind == de_AUT) && (destination == own_addr)) {flag_CTS =
false; flag_RTS = false;delay_counter = (delay_CTS + 10);State =1;flag_AUT = false;flag_ASS = false;}
                                        if ((msg_kind == de_ASS) && (State ==3) && (destination ==
own_addr)){ flag_CTS = false; flag_RTS = false;delay_counter = (delay_CTS + 10);flag_AUT = false;flag_ASS =
false;State =2;}
                                        }//if
                                if (msg_kind == CLK) {NAV = NAV - CLOCK; delay_counter++;if (NAV <= 0) NAV
= 0;}
                                }//receiving from CLK


                        //no station has tx during the SIFS
                        //this station will tx data
                        while (flag_CTS)
                                {//send out data packets

                                cMessage *work = new cMessage( "Data" );
                                work->setSCR(own_addr);
                                work->setDES(AP_addr);
                                //work->setROUTESCR(own_addr);
                                //work->setROUTDES(AP_addr);
                                work->setKind(DATA);
                                work->setNAV(0);
                                work->setMsgTime(simTime());
                                send( work, "out");


                                //wait for 1 clock cycle
                                flag_CLK = false;
                                while (!flag_CLK)
                                        {//receiving from CLK
                                        cMessage *done = receive();
                                        msg_kind = done->kind();
                                        delete done;
                                        if (msg_kind == CLK) {NAV = NAV - CLOCK; flag_CLK = true;}
                                        if (NAV <= 0) NAV = 0;
                                        }//receiving from CLK


                                if (NAV <= CLOCK) {flag_CTS = false;flag_DataSent = true;} //completed
sending all data less last packet

                                }//send out data packets

                                if (flag_DataSent)
                                {//flag_DataSent
                                        //send Last packet
                                        cMessage *work = new cMessage( "Last Pkt" );
                                        work->setSCR(own_addr);
                                        work->setDES(AP_addr);
                                        //work->setROUTESCR(own_addr);
                                        //work->setROUTDES(AP_addr);
                                        work->setKind(LASTPKT);
                                        work->setNAV(0);
                                        work->setMsgTime(simTime());
                                        send( work, "out");


                                        //recording vectors
                                        packet_count++;
                                        resp_v.record(packet_count); //xxx

                                        tx_time = simTime() - tx_stamp;
                                        //resp_b.record(tx_time);//xxx
```

```
                                        tx_stamp = simTime();

                                }//flag_DataSent

        }//if CTS



        //------ACK received----------
        if ((msg_kind == ACK) && (flag_RTS))
        {flag_RTS = false;} //no more outstanding RTS

                //delay of 1 cycle to allow other stations access to media
                flag_CLK = false;
                while (!flag_CLK)
                {//receiving from CLK
                cMessage *done = receive();
                msg_kind = done->kind();
                delete done;
                if (msg_kind == CLK) {NAV = NAV - CLOCK; flag_CLK = true;}
                if (NAV <= 0) NAV = 0;
                }//receiving from CLK



        }//if destination


        else

//--------------------PACKETs addressed to other station----------------------

        {//not to this destination


        // other station has asked for RTS and own has not sent RTS
        if ((msg_kind == RTS) && (!flag_RTS))
                {//if RTS
                if (msg_nav > NAV)
                        {//if msg_NAV
                        NAV = msg_nav;
                        time_stamp = simTime();
                        }//if msg_NAV
                flag_ORTS = true; // another Station has RTS
                flag_RTS = false; // back off

                }//if RTS


        // other station has asked for RTS, own has sent RTS
        if ((msg_kind == RTS) && (flag_RTS))
                {//if RTS
                if (msg_nav > NAV)
                        {//if msg_NAV
                        NAV = msg_nav;
                        time_stamp = simTime();
                        }//if msg_NAV
                flag_ORTS = true; // another Station has RTS
                flag_RTS = false; // back off

                }//if RTS




        }//not to this destination
```

```
//----------State 1 Authentication------------

                        if ((State ==1) && (flag_AUT == false))

                        {//if State ==1
                        //request for Authentication to AP
                        cMessage *work = new cMessage( "Auth" );
                        work->setSCR(own_addr);
                        work->setDES(AP_addr);
                        //work->setROUTESCR(own_addr);
                        //work->setROUTDES(AP_addr);
                        work->setKind(AUT);
                        work->setNAV(0);
                        work->setMsgTime(simTime());
                        NAV = 0;
                        time_stamp = simTime();
                        send( work, "out");
                        flag_AUT = true;  //oustanding AUT request
                        }//if State ==1


//----------State 2 Association------------

                        if ((State ==2) && (flag_ASS == false))
                        {//if State ==2
                        //request for Authentication to AP
                        cMessage *work = new cMessage( "Assoc" );
                        work->setSCR(own_addr);
                        work->setDES(AP_addr);
                        //work->setROUTESCR(own_addr);
                        //work->setROUTDES(AP_addr);
                        work->setKind(ASS);
                        work->setNAV(0);
                        work->setMsgTime(simTime());
                        send( work, "out");
                        flag_ASS = true;  //oustanding ASS request
                        time_stamp = simTime();
                        }//if State ==2



//----------State 3 Authenticated and Associated------------


                        if (State ==3)
                        {//if State == 3

                                if ((NAV == 0) && (!flag_RTS))
                                {//Send RTS

                                //RTS
                                cMessage *work = new cMessage( "RTS" );
                                work->setSCR(own_addr);
                                work->setDES(AP_addr);
                                //work->setROUTESCR(own_addr);
                                //work->setROUTDES(AP_addr);
                                work->setKind(RTS);
                                work->setNAV(data_len);
                                process_time = simTime();
                                work->setMsgTime(process_time);


                                flag_ORTS = false;//reset this flag

                                //Delay for DIFS
                                delay_counter = 1; // reset counter


                                //random number gen
```

```
                                        delay_len = intuniform(1, Device_Itvl, divresult.rem);

                                        delay_RTS = (DIFS / CLOCK)  + delay_len; //DIFS + x number of clock cycle

                                        while (delay_counter <= delay_RTS)// DIFS + x number of SIFS
                                        {//while delay
                                                cMessage *done = receive();
                                                destination = done->des_add();
                                                msg_kind = done->kind();
                                                msg_nav = done->nvec();
                                                delete done;
                                                //Clock Cycle received
                                                if (msg_kind == CLK)
                                                        {
                                                        if (NAV == 0) delay_counter++; //only increment if NAV = 0
                                                        NAV = NAV - CLOCK;
                                                        flag_ORTS = false;
                                                        if (NAV <= 0) NAV = 0;
                                                        }
                                                else
                                                        {//else
                                                        //if during any of the DIFS cycles is not a clock signal, then
check if it is an RTS, CTS, DeAut, DeASS
                                                        if (msg_nav > NAV) NAV = msg_nav;//set to the higher NAV
                                                        if (msg_kind == RTS) {flag_ORTS = true; delay_counter =
0;} //back off
                                                        if (msg_kind == CTS) {flag_ORTS = true; delay_counter =
0;} //back off
                                                        if ((msg_kind == de_AUT) && (destination == own_addr))
{State =1; flag_AUT = false;flag_ASS = false;delay_counter = (delay_RTS + 10);}
                                                        if ((msg_kind == de_ASS) && (State ==3) && (destination ==
own_addr)){ State =2;;flag_AUT = false;flag_ASS = false;delay_counter = (delay_RTS + 10);}
                                                        }//else


                                        }//while delay


                                        if (State == 3) //send RTS only if the station is at state 3 only
                                        {//if State 3
                                                //NAV Calculations for own RTS
                                                NAV = data_len;
                                                time_stamp = simTime();

                                                //Send the RTS message
                                                send( work, "out");
                                                flag_RTS = true;
                                        }//if State 3

                                        }// Send RTS

                                }//if State ==3


        } //for
        }//void
```

```
//---------------------------------------------------------------
// file: attacker.ccp
//       (part of WirelessLAN - an OMNeT++ simulation of IEEE802)
//---------------------------------------------------------------

#include "omnetpp.h"

class Attacker : public cSimpleModule
{
   Module_Class_Members(Attacker,cSimpleModule,16384)
   virtual void activity();
};

Define_Module( Attacker );

void Attacker::activity()
{

            int random_wait; // variance time randomised
            double process_time = 0;
            double time_stamp = 0;
            double elasped_time = 0;


            int data_len = 1000; // arbitual data transmission time


            //Addressing Information
            int own_addr = gate( "out" )->toGate()->index(); // sta own address
            int sta_add = parentModule()->par("num_station"); //total num of station
            int AP_addr = sta_add + 1; //permanent AP address
            int Attk_addr = sta_add + 2; // permanent addr for attacker
            const int data_addr = 99; //permanent addr for data sta

            //data parameters
            const int data_length = 8000; //1000 bytes (xxx Need to check on this one)

  //msg parameters
  int source; // msg source
  int destination; // msg destination
  int msg_kind; //msg kind
  int data_size; //length of data or
  long msg_nav; // nav of the msg
  float msg_timestamp; // time stamp of msg

  //IEEE802 vectors
  //--------------------------
  //State definitions
  int State = 1;
  //1 - Unauthenticated, Unassociated
  //2 - Authenticated, Unassociated
  //3 - Authenticated, Associated

  //Network Allocation Vector (NAV) - max value 32767 (appox 32 msec)

  long NAV = 0;
  long temp_NAV = 0; // updated NAV value of the incoming packet

  //Flags

  bool flag_AUT = false; // is there an outstanding AUT request
  bool flag_ASS = false; // is there an outstanding ASS request
  bool flag_CTS = false; // This station has CTS
  bool flag_RTS = false; // This station has RTS outstanding
  bool flag_ORTS = false;// Another Station has RTS outstanding
  bool flag_OCTS = false;// Another Station has CTS outstanding
  bool flag_CLK = false; // not waiting for CLK signal
```

```
  //Timing Constants - dependant on the PHY
                        //Device Interval - delays due to busy device
                        const int Device_Itvl = 3; // 3 cycles of clock

                        //DIFS
                        const int DIFS = 128; // microsecond
                        const int error_DIFS = 12; // microsecond

                        //SIFS
                        const int SIFS = 28; // microsecond
                        const int error_SIFS = 4; // microsecond

                        //Clock Cycle Time
                        const int CLOCK = 4; // microsecond

                        //microsecond denominator
                        const double Million = 0.000001;


                // Message Kind Definition
                const int RTS = 1;
                const int CTS = 2;
                const int AUT = 3;
                const int ASS = 4;
                const int rply_AUT = 5;
                const int rply_ASS = 6;
                const int de_AUT = 7;
                const int de_ASS = 8;
                const int DATA = 9;
                const int ACK = 10;
                const int CLK = 11;
                const int LASTPKT = 12;

//------------------------------------------------------------------------


                //clock cycle
                float delay = (CLOCK * Million);
 //------------------------------------------------------------------------


// do nothing
for(;;)
{//for



                //receiving from FS
                cMessage *done = receive();
                delete done;



} //for
}//void
```

```
//---------------------------------------------------------------
// file: attacker-DeAUT.cpp
//       (part of WirelessLAN - an OMNeT++ simulation of IEEE802)
//---------------------------------------------------------------

#include "omnetpp.h"

class Attacker : public cSimpleModule
{
   Module_Class_Members(Attacker,cSimpleModule,16384)
   virtual void activity();
};

Define_Module( Attacker );

void Attacker::activity()
{

            int random_wait; // variance time randomised
            double process_time = 0;
            double time_stamp = 0;
            double elasped_time = 0;


            int data_len = 1000; // arbitual data transmission time


            //Addressing Information
            int own_addr = gate( "out" )->toGate()->index(); // sta own address
            int sta_add = parentModule()->par("num_station"); //total num of station
            int AP_addr = sta_add + 1; //permanent AP address
            int Attk_addr = sta_add + 2; // permanent addr for attacker
            const int data_addr = 99; //permanent addr for data sta

            //data parameters
            const int data_length = 8000; //1000 bytes (xxx Need to check on this one)

  //msg parameters
  int source; // msg source
  int destination; // msg destination
  int msg_kind; //msg kind
  int data_size; //length of data or
  long msg_nav; // nav of the msg
  float msg_timestamp; // time stamp of msg



            //Attacker Parameters
  //--------------------

            bool flag_wait = false;
            const int Starttime = 3;
            int period = 3;
            int Endtime = Starttime + period;



  //IEEE802 vectors
  //--------------------------
  //State definitions
  int State = 1;
  //1 - Unauthenticated, Unassociated
  //2 - Authenticated, Unassociated
  //3 - Authenticated, Associated

  //Network Allocation Vector (NAV) - max value 32767 (appox 32 msec)

  long NAV = 0;
  long temp_NAV = 0; // updated NAV value of the incoming packet
```

```
  //Flags

  bool flag_AUT = false; // is there an outstanding AUT request
  bool flag_ASS = false; // is there an outstanding ASS request
  bool flag_CTS = false; // This station has CTS
  bool flag_RTS = false; // This station has RTS outstanding
  bool flag_ORTS = false;// Another Station has RTS outstanding
  bool flag_OCTS = false;// Another Station has CTS outstanding
  bool flag_CLK = false; // not waiting for CLK signal



  //Timing Constants - dependant on the PHY
                        //Device Interval - delays due to busy device
                        const int Device_Itvl = 3; // 3 cycles of clock

                        //DIFS
                        const int DIFS = 128; // microsecond
                        const int error_DIFS = 12; // microsecond

                        //SIFS
                        const int SIFS = 28; // microsecond
                        const int error_SIFS = 4; // microsecond

                        //Clock Cycle Time
                        const int CLOCK = 4; // microsecond

                        //microsecond denominator
                        const double Million = 0.000001;


            // Message Kind Definition
            const int RTS = 1;
            const int CTS = 2;
            const int AUT = 3;
            const int ASS = 4;
            const int rply_AUT = 5;
            const int rply_ASS = 6;
            const int de_AUT = 7;
            const int de_ASS = 8;
            const int DATA = 9;
            const int ACK = 10;
            const int CLK = 11;
            const int LASTPKT = 12;

//---------------------------------------------------------------------------


            //clock cycle
            float delay = (CLOCK * Million);
 //---------------------------------------------------------------------------


// deAUT station 1


for(;;)
{//for

            //conduct attack from Starttime to Endtime

            if (!flag_wait)
            {wait (Starttime); flag_wait = true;}


            if (simTime() < Endtime)
            {
            wait(0.000250);//Vary this value for different simulations
```

```
            //deAUT
            cMessage *work = new cMessage( "DeAUT" );
            work->setSCR(Attk_addr);
            work->setDES(1);
            work->setKind(de_AUT);
            work->setNAV(0);
            work->setMsgTime(simTime());
            send( work, "out");

            }

            else

            {cMessage *done = receive();
            delete done;
            }



} //for
}//void
```

```
//---------------------------------------------------------------
// file: attacker-DeAss.cpp
//       (part of WirelessLAN - an OMNeT++ simulation of IEEE802)
//---------------------------------------------------------------

#include "omnetpp.h"

class Attacker : public cSimpleModule
{
   Module_Class_Members(Attacker,cSimpleModule,16384)
   virtual void activity();
};

Define_Module( Attacker );

void Attacker::activity()
{

            int random_wait; // variance time randomised
            double process_time = 0;
            double time_stamp = 0;
            double elasped_time = 0;


            int data_len = 1000; // arbitual data transmission time


            //Addressing Information
            int own_addr = gate( "out" )->toGate()->index(); // sta own address
            int sta_add = parentModule()->par("num_station"); //total num of station
            int AP_addr = sta_add + 1; //permanent AP address
            int Attk_addr = sta_add + 2; // permanent addr for attacker
            const int data_addr = 99; //permanent addr for data sta

            //data parameters
            const int data_length = 8000; //1000 bytes (xxx Need to check on this one)

  //msg parameters
  int source; // msg source
  int destination; // msg destination
  int msg_kind; //msg kind
  int data_size; //length of data or
  long msg_nav; // nav of the msg
  float msg_timestamp; // time stamp of msg


  //Attacker Parameters
  //--------------------

            bool flag_wait = false;
            const int Starttime = 3;
            int period = 3;
            int Endtime = Starttime + period;


  //IEEE802 vectors
  //-------------------------
  //State definitions
  int State = 1;
  //1 - Unauthenticated, Unassociated
  //2 - Authenticated, Unassociated
  //3 - Authenticated, Associated

  //Network Allocation Vector (NAV) - max value 32767 (appox 32 msec)

  long NAV = 0;
  long temp_NAV = 0; // updated NAV value of the incoming packet

  //Flags
```

```
   bool flag_AUT = false; // is there an outstanding AUT request
   bool flag_ASS = false; // is there an outstanding ASS request
   bool flag_CTS = false; // This station has CTS
   bool flag_RTS = false; // This station has RTS outstanding
   bool flag_ORTS = false;// Another Station has RTS outstanding
   bool flag_OCTS = false;// Another Station has CTS outstanding
   bool flag_CLK = false; // not waiting for CLK signal


   //Timing Constants - dependant on the PHY
                        //Device Interval - delays due to busy device
                        const int Device_Itvl = 3; // 3 cycles of clock

                        //DIFS
                        const int DIFS = 128; // microsecond
                        const int error_DIFS = 12; // microsecond

                        //SIFS
                        const int SIFS = 28; // microsecond
                        const int error_SIFS = 4; // microsecond

                        //Clock Cycle Time
                        const int CLOCK = 4; // microsecond

                        //microsecond denominator
                        const double Million = 0.000001;


              // Message Kind Definition
              const int RTS = 1;
              const int CTS = 2;
              const int AUT = 3;
              const int ASS = 4;
              const int rply_AUT = 5;
              const int rply_ASS = 6;
              const int de_AUT = 7;
              const int de_ASS = 8;
              const int DATA = 9;
              const int ACK = 10;
              const int CLK = 11;
              const int LASTPKT = 12;

//---------------------------------------------------------------------------


              //clock cycle
              float delay = (CLOCK * Million);
 //---------------------------------------------------------------------------


// Dissassociated station 1


for(;;)
{//for



                        //conduct attack from Starttime to Endtime

              if (!flag_wait)
              {wait (Starttime); flag_wait = true;}


              if (simTime() < Endtime)
              {
              wait (0.000150); //vary this value for differnt runs
```

```
            //DeAss Stn 1
            cMessage *work = new cMessage( "DeASS" );
            work->setSCR(Attk_addr);
            work->setDES(1);
            work->setKind(de_ASS);
            work->setNAV(0);
            work->setMsgTime(simTime());
            send( work, "out");


            }

            else

            {cMessage *done = receive();
            delete done;
            }



} //for
}//void
```

```
//--------------------------------------------------------------
// file: attacker-RTS.cpp
//       (part of WirelessLAN - an OMNeT++ simulation of IEEE802)
//--------------------------------------------------------------

#include "omnetpp.h"

class Attacker : public cSimpleModule
{
   Module_Class_Members(Attacker,cSimpleModule,16384)
   virtual void activity();
};

Define_Module( Attacker );

void Attacker::activity()
{
            int random_wait; // variance time randomised
            double process_time = 0;
            double time_stamp = 0;
            double elasped_time = 0;


            int data_len = 1000; // arbitual data transmission time


            //Addressing Information
            int own_addr = gate( "out" )->toGate()->index(); // sta own address
            int sta_add = parentModule()->par("num_station"); //total num of station
            int AP_addr = sta_add + 1; //permanent AP address
            int Attk_addr = sta_add + 2; // permanent addr for attacker
            const int data_addr = 99; //permanent addr for data sta

            //data parameters
            const int data_length = 8000; //1000 bytes (xxx Need to check on this one)

  //msg parameters
  int source; // msg source
  int destination; // msg destination
  int msg_kind; //msg kind
  int data_size; //length of data or
  long msg_nav; // nav of the msg
  float msg_timestamp; // time stamp of msg


            //Attacker Parameters
  //--------------------

            bool flag_wait = false;
            int Starttime = 3;
            int Endtime = 6;



  //IEEE802 vectors
  //--------------------------
  //State definitions
  int State = 1;
  //1 - Unauthenticated, Unassociated
  //2 - Authenticated, Unassociated
  //3 - Authenticated, Associated

  //Network Allocation Vector (NAV) - max value 32767 (appox 32 msec)

  long NAV = 0;
  long temp_NAV = 0; // updated NAV value of the incoming packet

  //Flags
```

```
  bool flag_AUT = false; // is there an outstanding AUT request
  bool flag_ASS = false; // is there an outstanding ASS request
  bool flag_CTS = false; // This station has CTS
  bool flag_RTS = false; // This station has RTS outstanding
  bool flag_ORTS = false;// Another Station has RTS outstanding
  bool flag_OCTS = false;// Another Station has CTS outstanding
  bool flag_CLK = false; // not waiting for CLK signal



  //Timing Constants - dependant on the PHY
                        //Device Interval - delays due to busy device
                        const int Device_Itvl = 3; // 3 cycles of clock

                        //DIFS
                        const int DIFS = 128; // microsecond
                        const int error_DIFS = 12; // microsecond

                        //SIFS
                        const int SIFS = 28; // microsecond
                        const int error_SIFS = 4; // microsecond

                        //Clock Cycle Time
                        const int CLOCK = 4; // microsecond

                        //microsecond denominator
                        const double Million = 0.000001;


            // Message Kind Definition
            const int RTS = 1;
            const int CTS = 2;
            const int AUT = 3;
            const int ASS = 4;
            const int rply_AUT = 5;
            const int rply_ASS = 6;
            const int de_AUT = 7;
            const int de_ASS = 8;
            const int DATA = 9;
            const int ACK = 10;
            const int CLK = 11;
            const int LASTPKT = 12;

//---------------------------------------------------------------------------


            //clock cycle
            float delay = (CLOCK * Million);
 //---------------------------------------------------------------------------


//


for(;;)
{//for


            //conduct attack from Starttime to Endtime

            if (!flag_wait)
            {wait (Starttime); flag_wait = true;}


            if (simTime() < Endtime)
            {
            wait (0.002000);

            //RTS Attack
```

```
            cMessage *work = new cMessage( "RTS" );
            work->setSCR(Attk_addr);
            work->setDES(AP_addr);
            work->setKind(RTS);
            work->setNAV(1850);
            work->setMsgTime(simTime());
            send( work, "out");

            }//RTS Attack

            else

            {cMessage *done = receive();
            delete done;
            }


} //for
}//void
```

```
//--------------------------------------------------------------
// file: sta-DeAut.ccp
//       (part of WirelessLAN - an OMNeT++ simulation of IEEE802)
//--------------------------------------------------------------

#include "omnetpp.h"


class Station : public cSimpleModule
{
   Module_Class_Members(Station,cSimpleModule,16384)
   virtual void activity();
};

Define_Module( Station );

void Station::activity()
{
        float   process_time = 0;   // working variable for internal calculations
        float   time_stamp = 0;             // time stamp of NAV
        float   elasped_time = 0;   // working variable for internal calculations
        float   tx_time = 0;                // Time between successive packet transmissions
        float   tx_stamp = 0;               // Time stamp of packet sent, used in conjunction with
tx_time
        int             delay_len = 0;              // working variable for internal calculations
        int             delay_counter = 0;  // working variable for internal calculations
        int             true_counter = 0;
        int             packet_count = 0;
        int             delay_RTS;                          // random back off SIFS for RTS frame
        int             delay_CTS;                          // delay for CTS tx
        int             data_len = 120;             // arbitual data transmission time
        time_t  current_time;               // stores current pc clock time in seconds
        ldiv_t  divresult;                  // used for long division of integers in random num
generation


        //recording variables
        cOutVector resp_v("DataStation");       // time when data is transmitted
        cOutVector resp_b("Tx times");                  // time between successive transmissions

        //Addressing Information
        int             own_addr = gate( "out" )->toGate()->index();        //sta own address
        int             sta_add = parentModule()->par("num_station");   //total num of station
        int             AP_addr = sta_add + 1;
//permanent AP address
        int             Attk_addr = sta_add + 2;
//permanent addr for attacker
        const int data_addr = 99;
//permanent addr for data sta

        //data parameters
        const int data_length = 8000; //1000 bytes (xxx Need to check on this one)

  //msg parameters
  int           source;                 // msg source
  int           destination;        // msg destination
  int           msg_kind;               //msg kind
  long          msg_nav;            // nav of the msg
  float     msg_timestamp;      // time stamp of msg


  //IEEE802 vectors
  //-------------------------
  //State definitions
  int State = 1; //initial state
  //1 - Unauthenticated, Unassociated
  //2 - Authenticated, Unassociated
  //3 - Authenticated, Associated
```

```
  //Network Allocation Vector (NAV) - max value 32767 (appox 32 msec)

  long NAV = 0;
  long temp_NAV = 0; // updated NAV value of the incoming packet

  //Flags

  bool flag_AUT = false;        // is there an outstanding AUT request
  bool flag_ASS = false;        // is there an outstanding ASS request
  bool flag_CTS = false;        // This station has CTS
  bool flag_RTS = false;        // This station has RTS outstanding
  bool flag_ORTS = false;       // Another Station has RTS outstanding
  bool flag_OCTS = false;       // Another Station has CTS outstanding
  bool flag_CLK = false;        // not waiting for CLK signal
  bool flag_DataSent;                   // Data flag to indicate if there is a need to send the last packet
  bool flag_DeAUT = false;      // Indicate that a DeAUT is valid



  //Timing Constants - dependant on the PHY
                        //Device Interval - delays due to busy device
                        int Device_Itvl; // device delay

                        //DIFS
                        const int DIFS = 128;                   // microsecond
                        const int error_DIFS = 12;      // microsecond

                        //SIFS
                        const int SIFS = 28;            // microsecond
                        const int error_SIFS = 4;       // microsecond


                        //DeAUT, DeASS delay timeout
                        const int TimeOut = 500;        // microsecond

                        //Clock Cycle Time
                        const int CLOCK = 4;                    // microsecond

                        //microsecond denominator
                        const double Million = 0.000001;


            // Message Kind Definition
            const int RTS = 1;
            const int CTS = 2;
            const int AUT = 3;
            const int ASS = 4;
            const int rply_AUT = 5;
            const int rply_ASS = 6;
            const int de_AUT = 7;
            const int de_ASS = 8;
            const int DATA = 9;
            const int ACK = 10;
            const int CLK = 11;
            const int LASTPKT = 12;

//---------------------------------------------------------------------------




for(;;)
{//for

//randomly generate device interval for each cycle
current_time = time(NULL);
srand(current_time+own_addr);
divresult = ldiv (current_time,31);//the max possible RNG for intuniform is 31
```

```
Device_Itvl = intuniform(4, 7, divresult.rem);


//-----------------Incoming Pkt -------------------------


//receiving from FS
cMessage *done = receive();
source = done->scr_add();
destination = done->des_add();
msg_kind = done->kind();
msg_nav = done->nvec();
msg_timestamp = done->msgtimer();
delete done;



//-----------------------CLOCK Signal----------------------------
if (msg_kind == CLK)
{//if clock signal

        if (NAV > 0)
        {//if NAV >0
        NAV = NAV - CLOCK;
        if (NAV <=0) NAV = 0;
        }//if NAV >0



        time_stamp = simTime();

}//if clock signal



//---------------PACKETs addressed to this station----------------
if (destination == own_addr)
        {// if destination


//-------------------Authentication & Association------------------
        if ((msg_kind == rply_AUT) && (flag_AUT)) {State =2;flag_AUT = false; flag_ASS = false;}
        if ((msg_kind == rply_ASS) && (flag_ASS)) {State =3;flag_ASS = false;}

        if (msg_kind == de_AUT)

        {//if de_AUT (apply TimeOut and flag_DeAUT)
                //initialise
                delay_counter = TimeOut;
                flag_DeAUT = false;

                //sense the media to check for AP packets
                while ((delay_counter > 0) && (!flag_DeAUT))

                {//while

                        cMessage *done = receive();
                        source = done->scr_add();
                        destination = done->des_add();
                        msg_kind = done->kind();
                        msg_nav = done->nvec();
                        msg_timestamp = done->msgtimer();
                        delete done;

                        //---Clock---
                        if (msg_kind == CLK)
                        {
                        delay_counter = delay_counter - CLOCK;
                        NAV = NAV - CLOCK;
```

```
                                        if (delay_counter < 0) delay_counter = 0;
                                        }

                                        //---Check AP---
                                        if (source == AP_addr)
                                                {
                                                flag_DeAUT = true;
                                                if (msg_nav > NAV) NAV = msg_nav;
                                                }

                                }//while

                                //---Successful de_AUT---
                                if (!flag_DeAUT)
                                {
                                State =1;
                                flag_AUT = false;
                                flag_ASS = false;
                                }

                                //zerolise all parameters
                                msg_kind = 0;


                        }//if de_AUT



                        if ((msg_kind == de_ASS) && (State ==3)) State =2;


                //--------------------------CTS-----------------------------------

                        if ((msg_kind == CTS) && (State == 3) && (flag_RTS))
                        {//if CTS

                                //----NAV Calculuation----------
                                NAV = msg_nav; //set the NAV to that specified in CTS

                                flag_CTS = true; //CTS has been approved for this station
                                flag_DataSent = false; //no need to send last packet, only need if the first packets are sent


                                //delay 1 SIFS and listen out for other Transmitions
                                //if there is any transmission during the SIFS
                                //the CTS will be aborted (flag_CTS set to false)
                                flag_CLK = false;
                                delay_counter = 1;
                                delay_CTS = (SIFS / CLOCK)  + intrand(error_SIFS/CLOCK); //DIFS + x number of clock
cycle

                                while (delay_counter <= delay_CTS)// SIFS + x number of clock cycle
                                        {//while receiving from CLK
                                        cMessage *done = receive();
                                        destination = done->des_add();
                                        msg_kind = done->kind();
                                        msg_nav = done->nvec();
                                        delete done;
                                        if (msg_kind != CLK)
                                                {//if
                                                if (msg_kind == RTS) {flag_CTS = false; flag_RTS = false;NAV =
msg_nav; delay_counter = (delay_CTS + 10); if (NAV <= 0) NAV = 0;}//back off because someone else is transmitting
RTS
                                                if (msg_kind == CTS) {flag_CTS = false; flag_RTS = false;NAV =
msg_nav; delay_counter = (delay_CTS + 10); if (NAV <= 0) NAV = 0;}//back off because someone else is transmitting
CTS
                                                if (msg_kind == DATA) {flag_CTS = false; flag_RTS = false;NAV =
msg_nav; delay_counter = (delay_CTS + 10); if (NAV <= 0) NAV = 0;}//back off because someone else is transmitting
DATA
```

```
if ((msg_kind == de_AUT) && (destination == own_addr))
        {//if de_AUT (apply TimeOut and flag_DeAUT)
        //initialise
        delay_counter = TimeOut;
        flag_DeAUT = false;

        //sense the media to check for AP packets
        while ((delay_counter > 0) && (!flag_DeAUT))

        {//while

        cMessage *done = receive();
        source = done->scr_add();
        destination = done->des_add();
        msg_kind = done->kind();
        msg_nav = done->nvec();
        msg_timestamp = done->msgtimer();
        delete done;

        //---Clock---
        if (msg_kind == CLK)
        {
        delay_counter = delay_counter - CLOCK;
        NAV = NAV - CLOCK;
        if (delay_counter < 0) delay_counter = 0;
        }

        //---Check AP---
        if (source == AP_addr)
        {
        flag_DeAUT = true;
        if (msg_nav > NAV) NAV = msg_nav;
        }

        }//while

//---Successful de_AUT---
if (!flag_DeAUT)
{
flag_RTS = false;
State =1;
flag_AUT = false;
flag_ASS = false;
}

//zerolise all parameters
msg_kind = 0;
flag_CTS = false;
delay_counter = (delay_CTS + 10);

}//if de_AUT
```

```
        if ((msg_kind == de_ASS) && (State ==3) && (destination == own_addr)){ flag_CTS = false; flag_RTS = false;delay_counter = (delay_CTS + 10);flag_AUT = false;flag_ASS = false;State =2;}
        }//if
if (msg_kind == CLK) {NAV = NAV - CLOCK; delay_counter++;if (NAV <= 0) NAV = 0;}
}//while receiving from CLK
```

```
//no station has tx during the SIFS
//this station will tx data
while (flag_CTS)
	{//send out data packets

	cMessage *work = new cMessage( "Data" );
	work->setSCR(own_addr);
	work->setDES(AP_addr);
	//work->setROUTESCR(own_addr);
	//work->setROUTDES(AP_addr);
	work->setKind(DATA);
	work->setNAV(0);
	work->setMsgTime(simTime());
	send( work, "out");


	//wait for 1 clock cycle
	flag_CLK = false;
	while (!flag_CLK)
		{//receiving from CLK
		cMessage *done = receive();
		msg_kind = done->kind();
		delete done;
		if (msg_kind == CLK) {NAV = NAV - CLOCK; flag_CLK = true;}
		if (NAV <= 0) NAV = 0;
		}//receiving from CLK

	if (NAV <= CLOCK) {flag_CTS = false;flag_DataSent = true;} //completed
sending all data less last packet

	}//send out data packets

	if (flag_DataSent)
	{//flag_DataSent
		//send Last packet
		cMessage *work = new cMessage( "Last Pkt" );
		work->setSCR(own_addr);
		work->setDES(AP_addr);
		//work->setROUTESCR(own_addr);
		//work->setROUTDES(AP_addr);
		work->setKind(LASTPKT);
		work->setNAV(0);
		work->setMsgTime(simTime());
		send( work, "out");


		//recording vectors
		packet_count++;
		resp_v.record(packet_count); //xxx

		tx_time = simTime() - tx_stamp;
		//resp_b.record(tx_time);//xxx
		tx_stamp = simTime();

	}//flag_DataSent

}//if CTS



//------ACK received----------
if ((msg_kind == ACK) && (flag_RTS))
{flag_RTS = false;} //no more outstanding RTS

	//delay of 1 cycle to allow other stations access to media
	flag_CLK = false;
	while (!flag_CLK)
	{//receiving from CLK
	cMessage *done = receive();
	msg_kind = done->kind();
```

```
                delete done;
                if (msg_kind == CLK) {NAV = NAV - CLOCK; flag_CLK = true;}
                if (NAV <= 0) NAV = 0;
                }//receiving from CLK


        }//if destination


        else

//--------------------PACKETs addressed to other station----------------------

        {//not to this destination


        // other station has asked for RTS and own has not sent RTS
        if ((msg_kind == RTS) && (!flag_RTS))
                {//if RTS
                if (msg_nav > NAV)
                        {//if msg_NAV
                        NAV = msg_nav;
                        time_stamp = simTime();
                        }//if msg_NAV
                flag_ORTS = true; // another Station has RTS
                flag_RTS = false; // back off

                }//if RTS


        // other station has asked for RTS, own has sent RTS
        if ((msg_kind == RTS) && (flag_RTS))
                {//if RTS
                if (msg_nav > NAV)
                        {//if msg_NAV
                        NAV = msg_nav;
                        time_stamp = simTime();
                        }//if msg_NAV
                flag_ORTS = true; // another Station has RTS
                flag_RTS = false; // back off
                }//if RTS




        }//not to this destination


//----------State 1 Authentication------------

                        if ((State ==1) && (flag_AUT == false))

                        {//if State ==1
                        //request for Authentication to AP
                        cMessage *work = new cMessage( "Auth" );
                        work->setSCR(own_addr);
                        work->setDES(AP_addr);
                        //work->setROUTESCR(own_addr);
                        //work->setROUTDES(AP_addr);
                        work->setKind(AUT);
                        work->setNAV(0);
                        work->setMsgTime(simTime());
                        NAV = 0;
                        time_stamp = simTime();
                        send( work, "out");
                        flag_AUT = true;  //oustanding AUT request
                        }//if State ==1
```

```
//----------State 2 Association------------

                    if ((State ==2) && (flag_ASS == false))
                    {//if State ==2
                    //request for Authentication to AP
                    cMessage *work = new cMessage( "Assoc" );
                    work->setSCR(own_addr);
                    work->setDES(AP_addr);
                    //work->setROUTESCR(own_addr);
                    //work->setROUTDES(AP_addr);
                    work->setKind(ASS);
                    work->setNAV(0);
                    work->setMsgTime(simTime());
                    send( work, "out");
                    flag_ASS = true;  //oustanding ASS request
                    time_stamp = simTime();
                    }//if State ==2



//----------State 3 Authenticated and Associated------------


                    if (State ==3)
                    {//if State == 3

                              if ((NAV == 0) && (!flag_RTS))
                              {//Send RTS

                              //RTS
                              cMessage *work = new cMessage( "RTS" );
                              work->setSCR(own_addr);
                              work->setDES(AP_addr);
                              //work->setROUTESCR(own_addr);
                              //work->setROUTDES(AP_addr);
                              work->setKind(RTS);
                              work->setNAV(data_len);
                              process_time = simTime();
                              work->setMsgTime(process_time);


                              flag_ORTS = false;//reset this flag

                              //Delay for DIFS
                              delay_counter = 1; // reset counter


                              //random number gen
                              delay_len = intuniform(1, Device_Itvl, divresult.rem);

                              delay_RTS = (DIFS / CLOCK)  + delay_len; //DIFS + x number of clock cycle

                              while (delay_counter <= delay_RTS)// DIFS + x number of SIFS
                              {//while delay
                                        cMessage *done = receive();
                                        destination = done->des_add();
                                        msg_kind = done->kind();
                                        msg_nav = done->nvec();
                                        delete done;
                                        //Clock Cycle received
                                        if (msg_kind == CLK)
                                                  {
                                                  if (NAV == 0) delay_counter++; //only increment if NAV = 0
                                                  NAV = NAV - CLOCK;
                                                  flag_ORTS = false;
                                                  if (NAV <= 0) NAV = 0;
                                                  }
```

```
                                                    else
                                                            {//else
                                                            //if during any of the DIFS cycles is not a clock signal, then
check if it is an RTS, CTS, DeAut, DeASS

                                                            if (msg_nav > NAV) NAV = msg_nav;//set to the higher NAV
                                                            if (msg_kind == RTS) {flag_ORTS = true; delay_counter =
0;} //back off

                                                            if (msg_kind == CTS) {flag_ORTS = true; delay_counter =
0;} //back off



                                                            if ((msg_kind == de_AUT) && (destination == own_addr))
                                                                    {//if de_AUT (apply TimeOut and
flag_DeAUT)
                                                                    //initialise
                                                                    delay_counter = TimeOut;
                                                                    flag_DeAUT = false;

                                                                    //sense the media to check for AP
packets
                                                                    while ((delay_counter > 0) &&
(!flag_DeAUT))

                                                                    {//while

                                                                            cMessage *done = receive();
                                                                            source = done->scr_add();
                                                                            destination = done-
>des_add();

                                                                            msg_kind = done->kind();
                                                                            msg_nav = done->nvec();
                                                                            msg_timestamp = done-
>msgtimer();

                                                                            delete done;

                                                                            //---Clock---
                                                                            if (msg_kind == CLK)
                                                                            {
                                                                            delay_counter =
delay_counter - CLOCK;
                                                                            NAV = NAV - CLOCK;
                                                                            if (delay_counter < 0)
delay_counter = 0;
                                                                            }

                                                                            //---Check AP---
                                                                            if (source == AP_addr)
                                                                            {
                                                                                    flag_DeAUT = true;
                                                                                    if (msg_nav > NAV)
NAV = msg_nav;
                                                                            }

                                                                    }//while

                                                                    //---Successful de_AUT---
                                                                    if (!flag_DeAUT)
                                                                    {
                                                                    State =1;
                                                                    flag_AUT = false;
                                                                    flag_ASS = false;

                                                                    }

                                                                    //zerolise all parameters
                                                                    msg_kind = 0;
                                                                    delay_counter = (delay_RTS + 10);

                                                            }//if de_AUT
```

```
                                                    if ((msg_kind == de_ASS) && (State ==3) && (destination ==
own_addr)){ State =2;flag_AUT = false;flag_ASS = false;delay_counter = (delay_RTS + 10);}
                                                    }//else

                                        }//while delay


                                        if ((State == 3) && (NAV <=0)) //send RTS only if the station is at state 3 only
                                        {//if State 3
                                                //NAV Calculations for own RTS
                                                NAV = data_len;
                                                time_stamp = simTime();

                                                //Send the RTS message
                                                send( work, "out");
                                                flag_RTS = true;
                                        }//if State 3

                                        }// Send RTS


                                }//if State ==3



        } //for
        }//void
```

```
//---------------------------------------------------------------
// file: sta-DeASS.ccp
//       (part of WirelessLAN - an OMNeT++ simulation of IEEE802)
//---------------------------------------------------------------

#include "omnetpp.h"


class Station : public cSimpleModule
{
   Module_Class_Members(Station,cSimpleModule,16384)
   virtual void activity();
};

Define_Module( Station );

void Station::activity()
{

        float   process_time = 0;   // working variable for internal calculations
        float   time_stamp = 0;             // time stamp of NAV
        float   elasped_time = 0;   // working variable for internal calculations
        float   tx_time = 0;                // Time between successive packet transmissions
        float   tx_stamp = 0;               // Time stamp of packet sent, used in conjunction with
tx_time
        int             delay_len = 0;              // working variable for internal calculations
        int             delay_counter = 0;  // working variable for internal calculations
        int             true_counter = 0;
        int             packet_count = 0;
        int             delay_RTS;                          // random back off SIFS for RTS frame
        int             delay_CTS;                          // delay for CTS tx
        int             data_len = 120;             // arbitual data transmission time
        time_t  current_time;               // stores current pc clock time in seconds
        ldiv_t  divresult;                  // used for long division of integers in random num
generation


        //recording variables
        cOutVector resp_v("DataStation");       // time when data is transmitted
        cOutVector resp_b("Tx times");                  // time between successive transmissions

        //Addressing Information
        int             own_addr = gate( "out" )->toGate()->index();        //sta own address
        int             sta_add = parentModule()->par("num_station");   //total num of station
        int             AP_addr = sta_add + 1;
//permanent AP address
        int             Attk_addr = sta_add + 2;
//permanent addr for attacker
        const int data_addr = 99;
//permanent addr for data sta

        //data parameters
        const int data_length = 8000; //1000 bytes (xxx Need to check on this one)

  //msg parameters
  int           source;                 // msg source
  int           destination;        // msg destination
  int           msg_kind;               //msg kind
  long          msg_nav;            // nav of the msg
  float     msg_timestamp;      // time stamp of msg


  //IEEE802 vectors
  //--------------------------
  //State definitions
  int State = 1; //initial state
  //1 - Unauthenticated, Unassociated
  //2 - Authenticated, Unassociated
  //3 - Authenticated, Associated
```

```
  //Network Allocation Vector (NAV) - max value 32767 (appox 32 msec)

  long NAV = 0;
  long temp_NAV = 0; // updated NAV value of the incoming packet

  //Flags

  bool flag_AUT = false;        // is there an outstanding AUT request
  bool flag_ASS = false;        // is there an outstanding ASS request
  bool flag_CTS = false;        // This station has CTS
  bool flag_RTS = false;        // This station has RTS outstanding
  bool flag_ORTS = false;       // Another Station has RTS outstanding
  bool flag_OCTS = false;       // Another Station has CTS outstanding
  bool flag_CLK = false;        // not waiting for CLK signal
  bool flag_DataSent;                   // Data flag to indicate if there is a need to send the last packet
  bool flag_DeAUT = false;      // Indicate that a DeAUT is valid
  bool flag_DeASS = false;      // Indicate that a DeASS is vaild



  //Timing Constants - dependant on the PHY
                        //Device Interval - delays due to busy device
                        int Device_Itvl; // device delay

                        //DIFS
                        const int DIFS = 128;                   // microsecond
                        const int error_DIFS = 12;      // microsecond

                        //SIFS
                        const int SIFS = 28;            // microsecond
                        const int error_SIFS = 4;       // microsecond


                        //DeAUT, DeASS delay timeout
                        const int TimeOut = 500;        // microsecond

                        //Clock Cycle Time
                        const int CLOCK = 4;                    // microsecond

                        //microsecond denominator
                        const double Million = 0.000001;


            // Message Kind Definition
            const int RTS = 1;
            const int CTS = 2;
            const int AUT = 3;
            const int ASS = 4;
            const int rply_AUT = 5;
            const int rply_ASS = 6;
            const int de_AUT = 7;
            const int de_ASS = 8;
            const int DATA = 9;
            const int ACK = 10;
            const int CLK = 11;
            const int LASTPKT = 12;

//---------------------------------------------------------------------------




for(;;)
{//for

//randomly generate device interval for each cycle
current_time = time(NULL);
```

```
srand(current_time+own_addr);
divresult = ldiv (current_time,31);//the max possible RNG for intuniform is 31
Device_Itvl = intuniform(4, 7, divresult.rem);



//-----------------Incoming Pkt -------------------------


//receiving from FS
cMessage *done = receive();
source = done->scr_add();
destination = done->des_add();
msg_kind = done->kind();
msg_nav = done->nvec();
msg_timestamp = done->msgtimer();
delete done;



//----------------------CLOCK Signal---------------------------
if (msg_kind == CLK)
{//if clock signal

        if (NAV > 0)
        {//if NAV >0
        NAV = NAV - CLOCK;
        if (NAV <=0) NAV = 0;
        }//if NAV >0

        time_stamp = simTime();

}//if clock signal



//---------------PACKETs addressed to this station----------------
if (destination == own_addr)
        {// if destination


//------------------Authentication & Association------------------
        if ((msg_kind == rply_AUT) && (flag_AUT)) {State =2;flag_AUT = false; flag_ASS = false;}
        if ((msg_kind == rply_ASS) && (flag_ASS)) {State =3;flag_ASS = false;}


//------------------DeAuth & DeAss Handler------------------
        //--------------DeAuth--------------------
        if (msg_kind == de_AUT)

        {//if de_AUT (apply TimeOut and flag_DeAUT)
                //initialise
                delay_counter = TimeOut;
                flag_DeAUT = false;

                //sense the media to check for AP packets
                while ((delay_counter > 0) && (!flag_DeAUT))

                {//while

                        cMessage *done = receive();
                        source = done->scr_add();
                        destination = done->des_add();
                        msg_kind = done->kind();
                        msg_nav = done->nvec();
                        msg_timestamp = done->msgtimer();
                        delete done;

                        //---Clock---
                        if (msg_kind == CLK)
```

```
                {
                delay_counter = delay_counter - CLOCK;
                NAV = NAV - CLOCK;
                if (delay_counter < 0) delay_counter = 0;
                }

                //---Check AP---
                if (source == AP_addr)
                        {
                        flag_DeAUT = true;
                        if (msg_nav > NAV) NAV = msg_nav;
                        }

        }//while

        //---Successful de_AUT---
        if (!flag_DeAUT)
        {
        State =1;
        flag_AUT = false;
        flag_ASS = false;
        }

        //zerolise all parameters
        msg_kind = 0;


}//if de_AUT


//--------------DeASS--------------------
if ((msg_kind == de_ASS) && (State ==3))

{//if de_ASS (apply TimeOut and flag_DeASS)
        //initialise
        delay_counter = TimeOut;
        flag_DeASS = false;

        //sense the media to check for AP packets
        while ((delay_counter > 0) && (!flag_DeASS))

        {//while

                cMessage *done = receive();
                source = done->scr_add();
                destination = done->des_add();
                msg_kind = done->kind();
                msg_nav = done->nvec();
                msg_timestamp = done->msgtimer();
                delete done;

                //---Clock---
                if (msg_kind == CLK)
                {
                delay_counter = delay_counter - CLOCK;
                NAV = NAV - CLOCK;
                if (delay_counter < 0) delay_counter = 0;
                }

                //---Check AP---
                if (source == AP_addr)
                        {
                        flag_DeASS = true;
                        if (msg_nav > NAV) NAV = msg_nav;
                        }

        }//while

        //---Successful de_ASS---
        if (!flag_DeASS)
```

```
                        {
                        State =2;
                        flag_AUT = false;
                        flag_ASS = false;
                        }

                        //zerolise all parameters
                        msg_kind = 0;


                }//if de_ASS



        //---------------------------CTS-------------------------------------

                if ((msg_kind == CTS) && (State == 3) && (flag_RTS))
                {//if CTS

                        //----NAV Calculuation----------
                        NAV = msg_nav; //set the NAV to that specified in CTS

                        flag_CTS = true; //CTS has been approved for this station
                        flag_DataSent = false; //no need to send last packet, only need if the first packets are sent


                        //delay 1 SIFS and listen out for other Transmitions
                        //if there is any transmission during the SIFS
                        //the CTS will be aborted (flag_CTS set to false)
                        flag_CLK = false;
                        delay_counter = 1;
                        delay_CTS = (SIFS / CLOCK)  + intrand(error_SIFS/CLOCK); //DIFS + x number of clock
cycle

                        while (delay_counter <= delay_CTS)// SIFS + x number of clock cycle
                                {//while receiving from CLK
                                cMessage *done = receive();
                                destination = done->des_add();
                                msg_kind = done->kind();
                                msg_nav = done->nvec();
                                delete done;
                                if (msg_kind != CLK)
                                        {//if NOT CLOCK
                                        if (msg_kind == RTS) {flag_CTS = false; flag_RTS = false;NAV =
msg_nav; delay_counter = (delay_CTS + 10); if (NAV <= 0) NAV = 0;}//back off because someone else is transmitting
RTS
                                        if (msg_kind == CTS) {flag_CTS = false; flag_RTS = false;NAV =
msg_nav; delay_counter = (delay_CTS + 10); if (NAV <= 0) NAV = 0;}//back off because someone else is transmitting
CTS
                                        if (msg_kind == DATA) {flag_CTS = false; flag_RTS = false;NAV =
msg_nav; delay_counter = (delay_CTS + 10); if (NAV <= 0) NAV = 0;}//back off because someone else is transmitting
DATA




                                //------------DeAUT Handler-------------
                                if ((msg_kind == de_AUT) && (destination == own_addr))
                                        {//if de_AUT (apply TimeOut and flag_DeAUT)
                                        //initialise
                                        delay_counter = TimeOut;
                                        flag_DeAUT = false;

                                        //sense the media to check for AP packets
                                        while ((delay_counter > 0) && (!flag_DeAUT))

                                        {//while

                                        cMessage *done = receive();
                                        source = done->scr_add();
```

```
                destination = done->des_add();
                msg_kind = done->kind();
                msg_nav = done->nvec();
                msg_timestamp = done->msgtimer();
                delete done;

                //---Clock---
                if (msg_kind == CLK)
                {
                delay_counter = delay_counter - CLOCK;
                NAV = NAV - CLOCK;
                if (delay_counter < 0) delay_counter = 0;
                }

                //---Check AP---
                if (source == AP_addr)
                {
                flag_DeAUT = true;
                if (msg_nav > NAV) NAV = msg_nav;
                }

                }//while

        //---Successful de_AUT---
        if (!flag_DeAUT)
        {
        flag_RTS = false;
        State =1;
        flag_AUT = false;
        flag_ASS = false;
        }

        //zerolise all parameters
        msg_kind = 0;
        flag_CTS = false;
        delay_counter = (delay_CTS + 10);

        }//if de_AUT



        //--------------DeASS Handler---------------
        if ((msg_kind == de_ASS) && (State ==3) && (destination ==
own_addr))
                {//if de_ASS (apply TimeOut and flag_DeASS)
                //initialise
                delay_counter = TimeOut;
                flag_DeASS = false;

                //sense the media to check for AP packets
                while ((delay_counter > 0) && (!flag_DeASS))

                {//while

                        cMessage *done = receive();
                        source = done->scr_add();
                        destination = done->des_add();
                        msg_kind = done->kind();
                        msg_nav = done->nvec();
                        msg_timestamp = done->msgtimer();
                        delete done;

                        //---Clock---
                        if (msg_kind == CLK)
                        {
                        delay_counter = delay_counter - CLOCK;
                        NAV = NAV - CLOCK;
                        if (delay_counter < 0) delay_counter = 0;
```

```
                                                }

                                                //---Check AP---
                                                if (source == AP_addr)
                                                        {
                                                        flag_DeASS = true;
                                                        if (msg_nav > NAV) NAV = msg_nav;
                                                        }

                                        }//while

                                        //---Successful de_ASS---
                                        if (!flag_DeASS)
                                        {
                                        State =2;
                                        flag_AUT = false;
                                        flag_ASS = false;
                                        flag_CTS = false; flag_RTS = false;delay_counter =
(delay_CTS + 10);

                                        }

                                        //zerolise all parameters
                                        msg_kind = 0;


                                }//if de_ASS

                        }//if NOT CLOCK



                        if (msg_kind == CLK) {NAV = NAV - CLOCK; delay_counter++;if (NAV <= 0) NAV
= 0;}
                        }//while receiving from CLK


                //no station has tx during the SIFS
                //this station will tx data
                while (flag_CTS)
                        {//send out data packets

                        cMessage *work = new cMessage( "Data" );
                        work->setSCR(own_addr);
                        work->setDES(AP_addr);
                        //work->setROUTESCR(own_addr);
                        //work->setROUTDES(AP_addr);
                        work->setKind(DATA);
                        work->setNAV(0);
                        work->setMsgTime(simTime());
                        send( work, "out");


                        //wait for 1 clock cycle
                        flag_CLK = false;
                        while (!flag_CLK)
                                {//receiving from CLK
                                cMessage *done = receive();
                                msg_kind = done->kind();
                                delete done;
                                if (msg_kind == CLK) {NAV = NAV - CLOCK; flag_CLK = true;}
                                if (NAV <= 0) NAV = 0;
                                }//receiving from CLK

                        if (NAV <= CLOCK) {flag_CTS = false;flag_DataSent = true;} //completed
sending all data less last packet

                        }//send out data packets

                        if (flag_DataSent)
                        {//flag_DataSent
```

```
                                        //send Last packet
                                        cMessage *work = new cMessage( "Last Pkt" );
                                        work->setSCR(own_addr);
                                        work->setDES(AP_addr);
                                        //work->setROUTESCR(own_addr);
                                        //work->setROUTDES(AP_addr);
                                        work->setKind(LASTPKT);
                                        work->setNAV(0);
                                        work->setMsgTime(simTime());
                                        send( work, "out");


                                        //recording vectors
                                        packet_count++;
                                        resp_v.record(packet_count); //xxx

                                        tx_time = simTime() - tx_stamp;
                                        //resp_b.record(tx_time);//xxx
                                        tx_stamp = simTime();

                                }//flag_DataSent

        }//if CTS



        //------ACK received----------
        if ((msg_kind == ACK) && (flag_RTS))
        {flag_RTS = false;} //no more outstanding RTS

                //delay of 1 cycle to allow other stations access to media
                flag_CLK = false;
                while (!flag_CLK)
                {//receiving from CLK
                cMessage *done = receive();
                msg_kind = done->kind();
                delete done;
                if (msg_kind == CLK) {NAV = NAV - CLOCK; flag_CLK = true;}
                if (NAV <= 0) NAV = 0;
                }//receiving from CLK



        }//if destination


        else

//--------------------PACKETs addressed to other station----------------------

        {//not to this destination


        // other station has asked for RTS and own has not sent RTS
        if ((msg_kind == RTS) && (!flag_RTS))
                {//if RTS
                if (msg_nav > NAV)
                        {//if msg_NAV
                        NAV = msg_nav;
                        time_stamp = simTime();
                        }//if msg_NAV
                flag_ORTS = true; // another Station has RTS
                flag_RTS = false; // back off
                }//if RTS


        // other station has asked for RTS, own has sent RTS
        if ((msg_kind == RTS) && (flag_RTS))
                {//if RTS
                if (msg_nav > NAV)
```

```
                    {//if msg_NAV
                    NAV = msg_nav;
                    time_stamp = simTime();
                    }//if msg_NAV
                flag_ORTS = true; // another Station has RTS
                flag_RTS = false; // back off
                }//if RTS


        }//not to this destination


//----------State 1 Authentication------------

                if ((State ==1) && (flag_AUT == false))

                {//if State ==1
                //request for Authentication to AP
                cMessage *work = new cMessage( "Auth" );
                work->setSCR(own_addr);
                work->setDES(AP_addr);
                //work->setROUTESCR(own_addr);
                //work->setROUTDES(AP_addr);
                work->setKind(AUT);
                work->setNAV(0);
                work->setMsgTime(simTime());
                NAV = 0;
                time_stamp = simTime();
                send( work, "out");
                flag_AUT = true;  //oustanding AUT request
                }//if State ==1


//----------State 2 Association------------

                if ((State ==2) && (flag_ASS == false))
                {//if State ==2
                //request for Authentication to AP
                cMessage *work = new cMessage( "Assoc" );
                work->setSCR(own_addr);
                work->setDES(AP_addr);
                //work->setROUTESCR(own_addr);
                //work->setROUTDES(AP_addr);
                work->setKind(ASS);
                work->setNAV(0);
                work->setMsgTime(simTime());
                send( work, "out");
                flag_ASS = true;  //oustanding ASS request
                time_stamp = simTime();
                }//if State ==2



//----------State 3 Authenticated and Associated------------


                if (State ==3)
                {//if State == 3

                        if ((NAV == 0) && (!flag_RTS))
                        {//Send RTS

                        //RTS
                        cMessage *work = new cMessage( "RTS" );
                        work->setSCR(own_addr);
                        work->setDES(AP_addr);
                        //work->setROUTESCR(own_addr);
                        //work->setROUTDES(AP_addr);
```

```
work->setKind(RTS);
work->setNAV(data_len);
process_time = simTime();
work->setMsgTime(process_time);


flag_ORTS = false;//reset this flag

//Delay for DIFS
delay_counter = 1; // reset counter


//random number gen
delay_len = intuniform(1, Device_Itvl, divresult.rem);

delay_RTS = (DIFS / CLOCK)  + delay_len; //DIFS + x number of clock cycle

while (delay_counter <= delay_RTS)// DIFS + x number of SIFS
{//while delay
        cMessage *done = receive();
        destination = done->des_add();
        msg_kind = done->kind();
        msg_nav = done->nvec();
        delete done;
        //Clock Cycle received
        if (msg_kind == CLK)
                {
                if (NAV == 0) delay_counter++; //only increment if NAV = 0
                NAV = NAV - CLOCK;
                flag_ORTS = false;
                if (NAV <= 0) NAV = 0;
                }
        else
                {//else
                //if during any of the DIFS cycles is not a clock signal, then
check if it is an RTS, CTS, DeAut, DeASS
                if (msg_nav > NAV) NAV = msg_nav;//set to the higher NAV
                if (msg_kind == RTS) {flag_ORTS = true; delay_counter =
0;} //back off

                if (msg_kind == CTS) {flag_ORTS = true; delay_counter =
0;} //back off

                //---------------DeAUT Handler----------------

                if ((msg_kind == de_AUT) && (destination == own_addr))
                        {//if de_AUT (apply TimeOut and
flag_DeAUT)
                        //initialise
                        delay_counter = TimeOut;
                        flag_DeAUT = false;

                        //sense the media to check for AP
packets
                        while ((delay_counter > 0) &&
(!flag_DeAUT))

                        {//while

                                cMessage *done = receive();
                                source = done->scr_add();
                                destination = done-
>des_add();
                                msg_kind = done->kind();
                                msg_nav = done->nvec();
                                msg_timestamp = done-
>msgtimer();
                                delete done;

                                //---Clock---
                                if (msg_kind == CLK)
```

```
                                        {
                                        delay_counter                   =
delay_counter - CLOCK;

                                        NAV = NAV - CLOCK;
                                        if     (delay_counter     <     0)
delay_counter = 0;
                                        }

                                        //---Check AP---
                                        if (source == AP_addr)
                                        {
                                                flag_DeAUT = true;
                                                if (msg_nav > NAV)
NAV = msg_nav;
                                        }

                                }//while

                                //---Successful de_AUT---
                                if (!flag_DeAUT)
                                {
                                State =1;
                                flag_AUT = false;
                                flag_ASS = false;

                                }

                                //zerolise all parameters
                                msg_kind = 0;
                                delay_counter = (delay_RTS + 10);

                        }//if de_AUT


                //--------------De ASS handler-------------
                if ((msg_kind == de_ASS) && (State ==3) && (destination ==
own_addr))
                                                                {//if
de_ASS (apply TimeOut and flag_DeASS)
                //initialise
                delay_counter = TimeOut;
                flag_DeASS = false;

                //sense the media to check for AP packets
                while ((delay_counter > 0) && (!flag_DeASS))

                {//while

                        cMessage *done = receive();
                        source = done->scr_add();
                        destination = done->des_add();
                        msg_kind = done->kind();
                        msg_nav = done->nvec();
                        msg_timestamp = done->msgtimer();
                        delete done;

                        //---Clock---
                        if (msg_kind == CLK)
                        {
                        delay_counter = delay_counter - CLOCK;
                        NAV = NAV - CLOCK;
                        if (delay_counter < 0) delay_counter = 0;
                        }

                        //---Check AP---
                        if (source == AP_addr)
                                {
                                flag_DeASS = true;
                                if (msg_nav > NAV) NAV = msg_nav;
```

```
                                        }

                                }//while

                                //---Successful de_ASS---
                                if (!flag_DeASS)
                                {
                                State =2;
                                flag_AUT = false;
                                flag_ASS = false;
                                delay_counter = (delay_CTS + 10);
                                }

                                //zerolise all parameters
                                msg_kind = 0;


                        }//if de_ASS



                        }//else xxx


                }//while delay


                if ((State == 3) && (NAV <=0)) //send RTS only if the station is at state 3 only
                {//if State 3
                        //NAV Calculations for own RTS
                        NAV = data_len;
                        time_stamp = simTime();

                        //Send the RTS message
                        send( work, "out");
                        flag_RTS = true;
                }//if State 3

                }// Send RTS


        }//if State ==3



} //for
}//void
```

```
//--------------------------------------------------------------
// file: sta-protected.ccp
//       (part of WirelessLAN - an OMNeT++ simulation of IEEE802)
//--------------------------------------------------------------

#include "omnetpp.h"


class Station : public cSimpleModule
{
   Module_Class_Members(Station,cSimpleModule,16384)
   virtual void activity();
};

Define_Module( Station );

void Station::activity()
{
        float     process_time = 0;    // working variable for internal calculations
        float     time_stamp = 0;              // time stamp of NAV
        float     elasped_time = 0;    // working variable for internal calculations
        float     tx_time = 0;                 // Time between successive packet transmissions
        float     tx_stamp = 0;                // Time stamp of packet sent, used in conjunction with
tx_time
        int               delay_len = 0;               // working variable for internal calculations
        int               delay_counter = 0;  // working variable for internal calculations
        int               true_counter = 0;
        int               packet_count = 0;
        int               delay_RTS;                        // random back off SIFS for RTS frame
        int               delay_CTS;                        // delay for CTS tx
        int               counter_ORTS;             // counter when another Stn has asked for RTS
        int               data_len = 120;           // arbitual data transmission time
        time_t    current_time;             // stores current pc clock time in seconds
        ldiv_t    divresult;                // used for long division of integers in random num
generation


        //RECORDING VECTORS
        cOutVector resp_v("DataStation");       // time when data is transmitted
        cOutVector resp_b("Tx times");                  // time between successive transmissions

        //ADDRESSING INFORMATION
        int               own_addr = gate( "out" )->toGate()->index();      //sta own address
        int               sta_add = parentModule()->par("num_station");     //total num of station
        int               AP_addr = sta_add + 1;
//permanent AP address
        int               Attk_addr = sta_add + 2;
//permanent addr for attacker
        const int data_addr = 99;
//permanent addr for data sta

        //DATA PARAMETERS
        const int data_length = 8000; //1000 bytes (xxx Need to check on this one)

  //MSG PARAMETERS
  int           source;                   // msg source
  int           destination;        // msg destination
  int           msg_kind;                 //msg kind
  long          msg_nav;            // nav of the msg
  float     msg_timestamp;      // time stamp of msg


  //IEEE802 VECTORS
  //--------------------------
  //State definitions
  int State = 1; //initial state
  //1 - Unauthenticated, Unassociated
  //2 - Authenticated, Unassociated
  //3 - Authenticated, Associated
```

```
//Network Allocation Vector (NAV) - max value 32767 (appox 32 msec)

long NAV = 0;
long temp_NAV = 0; // updated NAV value of the incoming packet

//Flags

bool flag_AUT = false;        // is there an outstanding AUT request
bool flag_ASS = false;        // is there an outstanding ASS request
bool flag_CTS = false;        // This station has CTS
bool flag_RTS = false;        // This station has RTS outstanding
bool flag_ORTS = false;       // Another Station has RTS outstanding
bool flag_OCTS = false;       // Another Station has CTS outstanding
bool flag_CLK = false;        // not waiting for CLK signal
bool flag_DataSent;                   // Data flag to indicate if there is a need to send the last packet
bool flag_DeAUT = false;      // Indicate that a DeAUT is valid
bool flag_DeASS = false;      // Indicate that a DeASS is vaild


//TIMING CONSTANTS - dependant on the PHY

                    //Device Interval - delays due to busy device
                    int Device_Itvl; // device delay

                    //DIFS
                    const int DIFS = 128;                   // microsecond
                    const int error_DIFS = 12;      // microsecond

                    //SIFS
                    const int SIFS = 28;            // microsecond
                    const int error_SIFS = 4;       // microsecond


                    //DeAUT, DeASS delay timeout
                    const int TimeOut = 500;        // microsecond

                    //ORTS delay
                    const int delay_ORTS = 128;  // microsecond

                    //Clock Cycle Time
                    const int CLOCK = 4;                    // microsecond

                    //microsecond denominator
                    const double Million = 0.000001;


          // Message Kind Definition
          const int RTS = 1;
          const int CTS = 2;
          const int AUT = 3;
          const int ASS = 4;
          const int rply_AUT = 5;
          const int rply_ASS = 6;
          const int de_AUT = 7;
          const int de_ASS = 8;
          const int DATA = 9;
          const int ACK = 10;
          const int CLK = 11;
          const int LASTPKT = 12;

//-------------------------------------------------------------------------



for(;;)
```

```
{//for

//randomly generate device interval for each cycle
current_time = time(NULL);
srand(current_time+own_addr);
divresult = ldiv (current_time,31);//the max possible RNG for intuniform is 31
Device_Itvl = intuniform(4, 7, divresult.rem);



//-----------------Incoming Pkt -------------------------


//receiving from FS
cMessage *done = receive();
source = done->scr_add();
destination = done->des_add();
msg_kind = done->kind();
msg_nav = done->nvec();
msg_timestamp = done->msgtimer();
delete done;




//-----------------------CLOCK Signal----------------------------
if (msg_kind == CLK)
{//if clock signal

        if (NAV > 0)
        {//if NAV >0
        NAV = NAV - CLOCK;
        if (NAV <=0) NAV = 0;
        }//if NAV >0

        //Other Station has RTS handler
        if (flag_ORTS)
        {//ORTS
                counter_ORTS = counter_ORTS + CLOCK;
                if (counter_ORTS > delay_ORTS) {NAV = 0;flag_ORTS = false;} //no other transmission
during the del, reset the NAV to allow own RTS

        }//ORTS

}//if clock signal


//---------------PACKETs addressed to this station----------------
if (destination == own_addr)
        {// if destination


//------------------Authentication & Association------------------
        if ((msg_kind == rply_AUT) && (flag_AUT)) {State =2;flag_AUT = false; flag_ASS = false;}
        if ((msg_kind == rply_ASS) && (flag_ASS)) {State =3;flag_ASS = false;}


//------------------DeAuth & DeAss Handler------------------
        //-------------DeAuth--------------------
        if (msg_kind == de_AUT)

        {//if de_AUT (apply TimeOut and flag_DeAUT)
                //initialise
                delay_counter = TimeOut;
                flag_DeAUT = false;

                //sense the media to check for AP packets
                while ((delay_counter > 0) && (!flag_DeAUT))
```

```
		{//while

			cMessage *done = receive();
			source = done->scr_add();
			destination = done->des_add();
			msg_kind = done->kind();
			msg_nav = done->nvec();
			msg_timestamp = done->msgtimer();
			delete done;

			//---Clock---
			if (msg_kind == CLK)
			{
			delay_counter = delay_counter - CLOCK;
			NAV = NAV - CLOCK;
			if (delay_counter < 0) delay_counter = 0;
			}

			//---Check AP---
			if (source == AP_addr)
				{
				flag_DeAUT = true;
				if (msg_nav > NAV) NAV = msg_nav;
				}

		}//while

		//---Successful de_AUT---
		if (!flag_DeAUT)
		{
		State =1;
		flag_AUT = false;
		flag_ASS = false;
		}

		//zerolise all parameters
		msg_kind = 0;


}//if de_AUT


//--------------DeASS--------------------
if ((msg_kind == de_ASS) && (State ==3))

{//if de_ASS (apply TimeOut and flag_DeASS)
		//initialise
		delay_counter = TimeOut;
		flag_DeASS = false;

		//sense the media to check for AP packets
		while ((delay_counter > 0) && (!flag_DeASS))

		{//while

			cMessage *done = receive();
			source = done->scr_add();
			destination = done->des_add();
			msg_kind = done->kind();
			msg_nav = done->nvec();
			msg_timestamp = done->msgtimer();
			delete done;

			//---Clock---
			if (msg_kind == CLK)
			{
			delay_counter = delay_counter - CLOCK;
			NAV = NAV - CLOCK;
			if (delay_counter < 0) delay_counter = 0;
			}
```

```
                                        //---Check AP---
                                        if (source == AP_addr)
                                                {
                                                flag_DeASS = true;
                                                if (msg_nav > NAV) NAV = msg_nav;
                                                }

                                }//while

                                //---Successful de_ASS---
                                if (!flag_DeASS)
                                {
                                State =2;
                                flag_AUT = false;
                                flag_ASS = false;
                                }

                                //zerolise all parameters
                                msg_kind = 0;


                        }//if de_ASS



        //--------------------------CTS-----------------------------------

                        if ((msg_kind == CTS) && (State == 3) && (flag_RTS))
                        {//if CTS

                                //----NAV Calculuation----------
                                NAV = msg_nav; //set the NAV to that specified in CTS

                                flag_CTS = true; //CTS has been approved for this station
                                flag_DataSent = false; //no need to send last packet, only need if the first packets are sent


                                //delay 1 SIFS and listen out for other Transmitions
                                //if there is any transmission during the SIFS
                                //the CTS will be aborted (flag_CTS set to false)
                                flag_CLK = false;
                                delay_counter = 1;
                                delay_CTS = (SIFS / CLOCK)  + intrand(error_SIFS/CLOCK); //DIFS + x number of clock
cycle

                                while (delay_counter <= delay_CTS)// SIFS + x number of clock cycle
                                        {//while receiving from CLK
                                        cMessage *done = receive();
                                        source = done->scr_add();
                                        destination = done->des_add();
                                        msg_kind = done->kind();
                                        msg_nav = done->nvec();
                                        delete done;
                                        if (msg_kind != CLK)
                                                {//if NOT CLOCK
                                                if (msg_kind == RTS)
                                                {//back off because someone else is transmitting RTS
                                                        flag_CTS = false;
                                                        flag_RTS = false;
                                                        NAV = msg_nav;
                                                        delay_counter = (delay_CTS + 10);
                                                        if (NAV <= 0) NAV = 0;
                                                }//back off because someone else is transmitting RTS

                                                if (msg_kind == CTS) {flag_CTS = false; flag_RTS = false;NAV =
msg_nav; delay_counter = (delay_CTS + 10); if (NAV <= 0) NAV = 0;}//back off because someone else is transmitting
CTS
```

```
                                        if (msg_kind == DATA) {flag_CTS = false; flag_RTS = false;NAV =
msg_nav; delay_counter = (delay_CTS + 10); if (NAV <= 0) NAV = 0;}//back off because someone else is transmitting
DATA



                                        //------------DeAUT Handler-------------
                                        if ((msg_kind == de_AUT) && (destination == own_addr))
                                                {//if de_AUT (apply TimeOut and flag_DeAUT)
                                                //initialise
                                                delay_counter = TimeOut;
                                                flag_DeAUT = false;

                                                //sense the media to check for AP packets
                                                while ((delay_counter > 0) && (!flag_DeAUT))

                                                {//while

                                                cMessage *done = receive();
                                                source = done->scr_add();
                                                destination = done->des_add();
                                                msg_kind = done->kind();
                                                msg_nav = done->nvec();
                                                msg_timestamp = done->msgtimer();
                                                delete done;

                                                //---Clock---
                                                if (msg_kind == CLK)
                                                {
                                                delay_counter = delay_counter - CLOCK;
                                                NAV = NAV - CLOCK;
                                                if (delay_counter < 0) delay_counter = 0;
                                                }

                                                //---Check AP---
                                                if (source == AP_addr)
                                                {
                                                flag_DeAUT = true;
                                                if (msg_nav > NAV) NAV = msg_nav;
                                                }

                                                }//while

                                        //---Successful de_AUT---
                                        if (!flag_DeAUT)
                                        {
                                        flag_RTS = false;
                                        State =1;
                                        flag_AUT = false;
                                        flag_ASS = false;
                                        }

                                        //zerolise all parameters
                                        msg_kind = 0;
                                        flag_CTS = false;
                                        delay_counter = (delay_CTS + 10);

                                        }//if de_AUT




                                        //--------------DeASS Handler---------------
                                        if ((msg_kind == de_ASS) && (State ==3) && (destination ==
own_addr))
                                                {//if de_ASS (apply TimeOut and flag_DeASS)
                                                //initialise
                                                delay_counter = TimeOut;
```

```
                                        flag_DeASS = false;

                                        //sense the media to check for AP packets
                                        while ((delay_counter > 0) && (!flag_DeASS))

                                        {//while

                                                cMessage *done = receive();
                                                source = done->scr_add();
                                                destination = done->des_add();
                                                msg_kind = done->kind();
                                                msg_nav = done->nvec();
                                                msg_timestamp = done->msgtimer();
                                                delete done;

                                                //---Clock---
                                                if (msg_kind == CLK)
                                                {
                                                delay_counter = delay_counter - CLOCK;
                                                NAV = NAV - CLOCK;
                                                if (delay_counter < 0) delay_counter = 0;
                                                }

                                                //---Check AP---
                                                if (source == AP_addr)
                                                        {
                                                        flag_DeASS = true;
                                                        if (msg_nav > NAV) NAV = msg_nav;
                                                        }

                                        }//while

                                        //---Successful de_ASS---
                                        if (!flag_DeASS)
                                        {
                                        State =2;
                                        flag_AUT = false;
                                        flag_ASS = false;
                                        flag_CTS = false; flag_RTS = false;delay_counter =
(delay_CTS + 10);
                                        }

                                        //zerolise all parameters
                                        msg_kind = 0;


                                }//if de_ASS

                        }//if NOT CLOCK



                        if (msg_kind == CLK) {NAV = NAV - CLOCK; delay_counter++;if (NAV <= 0) NAV
= 0;}
                        }//while receiving from CLK


                //no station has tx during the SIFS
                //this station will tx data
                while (flag_CTS)
                        {//send out data packets

                        cMessage *work = new cMessage( "Data" );
                        work->setSCR(own_addr);
                        work->setDES(AP_addr);
                        //work->setROUTESCR(own_addr);
                        //work->setROUTDES(AP_addr);
                        work->setKind(DATA);
                        work->setNAV(0);
                        work->setMsgTime(simTime());
```

```
                                        send( work, "out");


                                        //wait for 1 clock cycle
                                        flag_CLK = false;
                                        while (!flag_CLK)
                                                {//receiving from CLK
                                                cMessage *done = receive();
                                                msg_kind = done->kind();
                                                delete done;
                                                if (msg_kind == CLK) {NAV = NAV - CLOCK; flag_CLK = true;}
                                                if (NAV <= 0) NAV = 0;
                                                }//receiving from CLK

                                        if (NAV <= CLOCK) {flag_CTS = false;flag_DataSent = true;} //completed
sending all data less last packet

                                        }//send out data packets

                                        if (flag_DataSent)
                                        {//flag_DataSent
                                                //send Last packet
                                                cMessage *work = new cMessage( "Last Pkt" );
                                                work->setSCR(own_addr);
                                                work->setDES(AP_addr);
                                                //work->setROUTESCR(own_addr);
                                                //work->setROUTDES(AP_addr);
                                                work->setKind(LASTPKT);
                                                work->setNAV(0);
                                                work->setMsgTime(simTime());
                                                send( work, "out");


                                                //recording vectors
                                                packet_count++;
                                                resp_v.record(packet_count); //xxx

                                                tx_time = simTime() - tx_stamp;
                                                //resp_b.record(tx_time);//xxx
                                                tx_stamp = simTime();

                                        }//flag_DataSent

                        }//if CTS



                        //------ACK received----------
                        if ((msg_kind == ACK) && (flag_RTS))
                        {flag_RTS = false;} //no more outstanding RTS

                                //delay of 1 cycle to allow other stations access to media
                                flag_CLK = false;
                                while (!flag_CLK)
                                {//receiving from CLK
                                cMessage *done = receive();
                                msg_kind = done->kind();
                                delete done;
                                if (msg_kind == CLK) {NAV = NAV - CLOCK; flag_CLK = true;}
                                if (NAV <= 0) NAV = 0;
                                }//receiving from CLK



                        }//if destination


                        else
```

```
//--------------------PACKETs addressed to other station----------------------

        {//not to this destination


        // other station has asked for RTS and own has not sent RTS
        if ((msg_kind == RTS) && (!flag_RTS))
                {//if RTS
                if (msg_nav > NAV)
                        {//if msg_NAV
                        NAV = msg_nav;
                        time_stamp = simTime();
                        }//if msg_NAV
                flag_ORTS = true; // another Station has RTS
                counter_ORTS = 0; //reset counter
                flag_RTS = false; // back off
                }//if RTS


        // other station has asked for RTS, own has sent RTS
        if ((msg_kind == RTS) && (flag_RTS))
                {//if RTS
                if (msg_nav > NAV)
                        {//if msg_NAV
                        NAV = msg_nav;
                        time_stamp = simTime();
                        }//if msg_NAV
                flag_ORTS = true; // another Station has RTS
                counter_ORTS = 0; //reset counter
                flag_RTS = false; // back off
                }//if RTS


        //xxxx need to add a CTS handler here xxx
        // AP respond CTS to other station
        if (msg_kind == CTS)
                {//if CTS
                if (msg_nav > NAV)
                        {//if msg_NAV
                        NAV = msg_nav;
                        time_stamp = simTime();
                        }//if msg_NAV
                flag_ORTS = true; // another Station has CTS
                counter_ORTS = 0; //reset counter
                flag_RTS = false; // back off
                }//if CTS

        //any other msg received will reset the ORTS flag
        if ((msg_kind != CTS)&&(msg_kind != CLK) && (msg_kind != RTS))
        {//if not CTS or CLK
                if (flag_ORTS) {flag_ORTS = false;NAV = 0;}

        }//if not CTS or CLK

        }//not to this destination



//----------State 1 Authentication------------

                if ((State ==1) && (flag_AUT == false))

                {//if State ==1
                //request for Authentication to AP
                cMessage *work = new cMessage( "Auth" );
                work->setSCR(own_addr);
                work->setDES(AP_addr);
                //work->setROUTESCR(own_addr);
                //work->setROUTDES(AP_addr);
                work->setKind(AUT);
```

```
work->setNAV(0);
work->setMsgTime(simTime());
NAV = 0;
time_stamp = simTime();
send( work, "out");
flag_AUT = true;  //oustanding AUT request
}//if State ==1

//----------State 2 Association-----------

if ((State ==2) && (flag_ASS == false))
{//if State ==2
//request for Authentication to AP
cMessage *work = new cMessage( "Assoc" );
work->setSCR(own_addr);
work->setDES(AP_addr);
//work->setROUTESCR(own_addr);
//work->setROUTDES(AP_addr);
work->setKind(ASS);
work->setNAV(0);
work->setMsgTime(simTime());
send( work, "out");
flag_ASS = true;  //oustanding ASS request
time_stamp = simTime();
}//if State ==2



//----------State 3 Authenticated and Associated-----------

if (State ==3)
{//if State == 3

    if ((NAV == 0) && (!flag_RTS))
    {//Send RTS

    //RTS
    cMessage *work = new cMessage( "RTS" );
    work->setSCR(own_addr);
    work->setDES(AP_addr);
    //work->setROUTESCR(own_addr);
    //work->setROUTDES(AP_addr);
    work->setKind(RTS);
    work->setNAV(data_len);
    process_time = simTime();
    work->setMsgTime(process_time);


    flag_ORTS = false;//reset this flag

    //Delay for DIFS
    delay_counter = 1; // reset counter


    //random number gen
    delay_len = intuniform(1, Device_Itvl, divresult.rem);

    delay_RTS = (DIFS / CLOCK)  + delay_len; //DIFS + x number of clock cycle

    while (delay_counter <= delay_RTS)// DIFS + x number of SIFS
    {//while delay

        cMessage *done = receive();
        source = done->scr_add();
        destination = done->des_add();
        msg_kind = done->kind();
        msg_nav = done->nvec();
```

```
delete done;
//Clock Cycle received
if (msg_kind == CLK)
        {
        if (NAV == 0) delay_counter++; //only increment if NAV = 0
        NAV = NAV - CLOCK;
        if (NAV <= 0) NAV = 0;
        }
else
        {//else
        //if during any of the DIFS cycles is not a clock signal, then
check if it is an RTS, CTS, DeAut, DeASS

        delay_counter = 1; // reset the counter because there is a tx
during DIFS and it is not CLK

        if (msg_nav > NAV) NAV = msg_nav;//set to the higher NAV

        if (msg_kind == RTS)
        {flag_ORTS = true; counter_ORTS = 0; delay_counter =
(delay_RTS + 10);} //back off due to other station RTS

        if (msg_kind == CTS)
        {flag_OCTS = true; counter_ORTS = 0; delay_counter =
(delay_RTS + 10);} //back off due to other station CTS


        //---------------DeAUT Handler----------------

        if ((msg_kind == de_AUT) && (destination == own_addr))
                {//if de_AUT (apply TimeOut and
flag_DeAUT)
                //initialise
                delay_counter = TimeOut;
                flag_DeAUT = false;

                //sense the media to check for AP
packets
                while ((delay_counter > 0) &&
(!flag_DeAUT))

                {//while

                        cMessage *done = receive();
                        source = done->scr_add();
                        destination = done-
>des_add();

                        msg_kind = done->kind();
                        msg_nav = done->nvec();
                        msg_timestamp = done-
>msgtimer();

                        delete done;

                        //---Clock---
                        if (msg_kind == CLK)
                        {
                        delay_counter =
delay_counter - CLOCK;

                        NAV = NAV - CLOCK;
                        if (delay_counter < 0)
delay_counter = 0;

                        }

                        //---Check AP---
                        if (source == AP_addr)
                        {
                                flag_DeAUT = true;
                                if (msg_nav > NAV)
NAV = msg_nav;

                        }
```

```
                    }//while

                    //---Successful de_AUT---
                    if (!flag_DeAUT)
                    {
                    State =1;
                    flag_AUT = false;
                    flag_ASS = false;

                    }

                    //zerolise all parameters
                    msg_kind = 0;
                    delay_counter = (delay_RTS + 10);

          }//if de_AUT



//--------------De ASS handler-------------
if ((msg_kind == de_ASS) && (State ==3) && (destination ==
```

own_addr))

de_ASS (apply TimeOut and flag_DeASS)

```
                                                    {//if

//initialise
delay_counter = TimeOut;
flag_DeASS = false;

//sense the media to check for AP packets
while ((delay_counter > 0) && (!flag_DeASS))

{//while

          cMessage *done = receive();
          source = done->scr_add();
          destination = done->des_add();
          msg_kind = done->kind();
          msg_nav = done->nvec();
          msg_timestamp = done->msgtimer();
          delete done;

          //---Clock---
          if (msg_kind == CLK)
          {
          delay_counter = delay_counter - CLOCK;
          NAV = NAV - CLOCK;
          if (delay_counter < 0) delay_counter = 0;
          }

          //---Check AP---
          if (source == AP_addr)
                    {
                    flag_DeASS = true;
                    if (msg_nav > NAV) NAV = msg_nav;
                    }

}//while

//---Successful de_ASS---
if (!flag_DeASS)
{
State =2;
flag_AUT = false;
flag_ASS = false;
delay_counter = (delay_CTS + 10);
}

//zerolise all parameters
```

```
                                        msg_kind = 0;


                                    }//if de_ASS



                                    }//else

                                }//while delay


                                if ((State == 3) && (NAV <=0)) //send RTS only if the station is at state 3 only
                                {//if State 3
                                    //NAV Calculations for own RTS
                                    NAV = data_len;
                                    time_stamp = simTime();

                                    //Send the RTS message
                                    send( work, "out");
                                    flag_RTS = true;
                                }//if State 3

                                }// Send RTS


                            }//if State ==3

} //for
}//void
```

LIST OF REFERENCES

[1] Nikita Borisov, Ian Goldberg, and David Wagner. Intercepting Mobile Communications: The Insecurity of 802.11. In Seventh Annual International Conference on Mobile Computing And Networking, Rome, Italy, July 2001.

[2] W.A. Arbaugh, N. Shankar, J.Wang, and K. Zhang. Your 802.11 Network has No Clothes. In First IEEE International Conference on Wireless LANs and Home Networks, Suntec City, Singapore, December 2001.

[3] Bellardo, John, and Savage, Stefan. “802.11 Denial-of-Service Attacks: Real Vulnerabilities and Practical Solutions.” Proceedings of the USENIX Security Symposium, August 2003.

[4] The Institute of Electrical and Electronics Engineers, Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications, 20 August 1999.

[5] O’Hara, B. and Petrick, A. The IEEE 802.11 Handbook: A Designer’s Companion, IEEE Press, 1999.

[6] András Varga. OMNeT++ Discrete Event Simulation System Version 2.3 User Manual. http://www.omnetpp.org. Accessed 15 November 2003.

[7] W.A. Arbaugh, N. Shankar, J.Wang, and K. Zhang. Your 802.11 Network has No Clothes. In First IEEE International Conference on Wireless LANs and Home Networks, Suntec City, Singapore, December 2001.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Dr William Ray
   Naval Postgraduate School
   Monterey, California

4. Dr Man-Tak Shing
   Naval Postgraduate School
   Monterey, California

5. Temasek Defence Systems Institute
   National University of Singapore
   Singapore

6. MAJ Boon Hwee Tan
   Republic of Singapore Navy
   Singapore